REDES NEURONALES CONVOLUCIONALES USANDO KERAS Y ACELERANDO CON GPU

Nelson Enrique Vera-Parra
Andrés Ovidio Restrepo-Rodríguez
Rubén Javier Medina-Daza

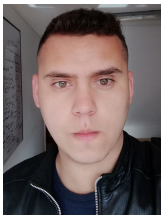
UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Doctorado en Ingeniería
UNIVERSIDAD DISTRITAL "FRANCISCO JOSÉ DE CALDAS"

Nelson Enrique Vera-Parra

Ingeniero Electrónico de la Universidad Surcolombiana, Magíster en Ciencias de la Información y las Comunicaciones y Doctor en Ingeniería de la Universidad Distrital Francisco José de Caldas. Profesor Titular de la misma Universidad. Investigador en HPC, Ciencia de datos y Bioinformática.

Andrés Ovidio Restrepo-Rodríguez



Ingeniero de sistemas y actual estudiante de la Maestría en Ciencias de la Información y las Comunicaciones en la Universidad Distrital Francisco José de Caldas. Mis campos de investigación incluyen Inteligencia Artificial, Learning Analytics, Entornos Inmersivos y Computación Heterogénea.

Rubén Javier Medina-Daza

Doctor en Informática, énfasis: Sistemas de Información Geográfica, Magíster en Teleinformática, Licenciado en Matemáticas de la Universidad Distrital Francisco José de Caldas. Profesor Titular de Ingeniería Catastral y Geodesia, de la Maestría en Ciencias de la Información y las Comunicaciones y del Doctorado de Ingeniería.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Doctorado en Ingeniería
UNIVERSIDAD DISTRITAL "FRANCISCO JOSÉ DE CALDAS"

REDES NEURONALES CONVOLUCIONALES USANDO KERAS Y ACELERANDO CON GPU

Nelson Enrique Vera-Parra
Andrés Ovidio Restrepo-Rodríguez
Rubén Javier Medina-Daza

Vera Parra, Nelson Enrique
Redes neuronales convolucionales usando keras y acelerando con GPU / Nelson Enrique Vera Parra, Andrés Ovidio Restrepo Rodríguez, Rubén Javier Medina Daza. -- 1a. ed. -- Bogotá : Universidad Distrital Francisco José de Caldas, 2020.

122 páginas ; 24 cm. -- (Doctorado en Ingeniería).

Incluye bibliografía.

ISBN 978-958-787-232-3 (impreso) -- 978-958-787-233-0 (digital)

1. Redes neuronales (computadores) I. Restrepo Rodríguez, Andrés Ovidio II. Medina Daza, Rubén Javier III. Título IV. Serie

CDD: 006.32 ed. 23 CO-BoBN– a1057507

ISBN Impreso: 978-958-787-232-3
ISBN Digital: 978-958-787-233-0

Primera edición: Bogotá, octubre de 2020.

Corrección de estilo y diseño gráfico:
Amadgraf Impresores Ltda.
Impresión:
Amadgraf Impresores Ltda.

Doctorado en Ingeniería
Carrera 7 # 40B-53
Bogotá
Correo electrónico: investigacion.doctoradoing@udistrital.edu.co

Hecho el depósito legal.

Impreso y hecho en Colombia

Tabla de Contenido

Índice de Ilustraciones

Índice de Tablas

Índice de Ecuaciones

Prefacio

Vivimos en una era gobernada por datos, donde la intuición y el azar se han visto rezagados ante predicciones que soportan tanto decisiones cotidianas como grandes políticas gubernamentales. Una era donde la nueva riqueza se encuentra en los datos y en los métodos que permiten hacer un uso eficiente de éstos. En los últimos años, los métodos de procesamiento de datos que mayor precisión han presentado en tareas predictivas han sido aquellos basados en aprendizaje profundo, como por ejemplo las redes neuronales convolucionales. Este tipo de métodos representan un reto tanto por su alta exigencia de recurso computacional como por su complejidad de diseño e implementación.

Tomando como motivación lo anterior, en este libro el lector encontrará una guía práctica para la implementación, entrenamiento y validación de redes neuronales convolucionales usando Keras y acelerando con GPU. La guía se desarrolla mediante un caso de estudio típico enmarcado en las clasificaciones de imágenes satelitales. Adicionalmente la evaluación del modelo implementado incluye la comparación a nivel de speed-up con los modelos de redes neuronales pre-entenados más comunes: MobileNet, MobileNetV2, ResNet50 y VGG16.

Introducción

En esta sección, se presentan conceptos claves y fundamentales referentes a aprendizaje de máquina, aprendizaje profundo de máquina y procesamiento de datos en arquitecturas heterógeneas CPU-GPU. Lo anterior, con el propósito de contextualizar al lector y brindar herramientas para la compresión de este libro.

¿Qué es el aprendizaje de máquina?

El término "aprender" en el dominio de las máquinas hace referencia a generalizar comportamientos a partir de una información suministrada en forma de ejemplos. De acuerdo a la forma que "aprenden" las máquinas, los algoritmos se puede clasificar en supervisados y no supervisados. En el aprendizaje supervisado los datos traen relacionado un objetivo, mientras que en el no supervisado los datos no traen ninguna relación explicita con algún objetivo. Los algoritmos que utilizan aprendizaje supervisado (vecinos más cercanos, máquinas de vectores de soporte, árboles de decisión, regresión, entre otros) normalmente cumplen funciones de clasificación o regresión, mientras que los no supervisados (modelos gausianos, aprendizaje múltiple, estimación de densidad, entre otros) cumplen funciones de agrupamiento.

¿Cuáles son las técnicas más comunes de aprendizaje de máquina supervisado?

Vecinos más cercanos (K-NN). Es un método de clasificación supervisada no paramétrico que permite estimar la función de densidad de probabilidad o la probabilidad a posteriori de que un elemento pertenece a una clase determinada. El cálculo de esta probabilidad se basa en analizar a qué clase pertenecen los k vecinos más cercanos. Para determinar cuáles son los vecinos más cercanos se utiliza generalmente la distancia euclidiana. Un buen ejemplo para esclarecer el funcionamiento de K-NN es su uso en la clasificación de clientes de bancos en confiables o no para realizarles un crédito. En la figura 1 el cliente a ser clasificado se denota con el cuadro verde, mientras que los clientes que han realizado créditos y no los han pagado con la estrella roja y aquellos que si pagaron con el triángulo azul. Si se emplea un k = 5, sería más probable que el cliente pague debido a que 3 de sus 5 vecinos también pagaron. Si se emplea un k = 10, sería más probable que el cliente no pague debido a que 6 de sus 10 vecinos tampoco pagaron (Wu Jian et.al., 2014).

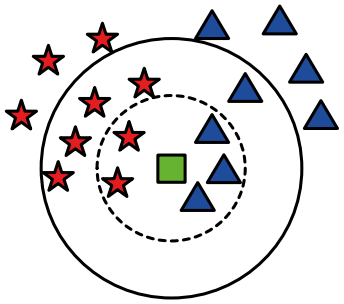


Figura 1 Ejemplo de clasificación de clientes
Fuente (Wu Jian et. al., 2014)

Máquinas de Vectores de Soporte (SVM). Técnica de aprendizaje supervisado creada por Vladimir Vapnik (1995) que permite realizar tareas de clasificación y de regresión mediante la creación de hiperplanos que separan los datos de entrada.

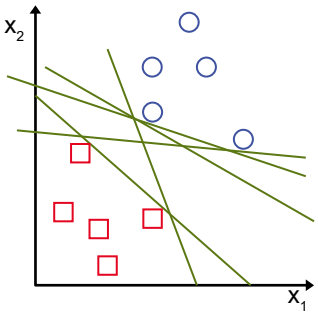


Figura 2 Hiperplano que separan los datos de entrada.
Fuente: http://www.swarthmore.edu/NatSci/mzucker1/opencv-2.4.10-docs/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html

Un hiperplano es un plano de n-1 dimensiones que divide en dos a un plano n dimensional. En la figura 2 se muestran algunos hiperplanos (en este caso rectas) que dividen en dos el plano bidimensional donde se encuentran unos datos de entrenamiento (cuadros y círculos). SVM permite encontrar el hiperplano que mejor separe los datos de entrenamiento, es decir aquel que presente una mayor margen hacia los datos de cada una de las clases (Paul Mather, Brandt Tso, 2009) (ver figura 3).

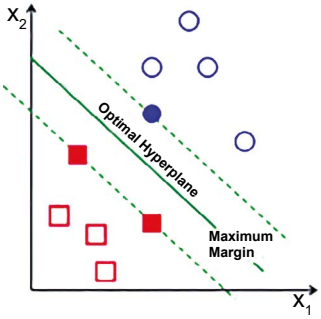


Figura 3 Hiperplano óptimo.
Fuente: http://www.swarthmore.edu/NatSci/mzucker1/opencv-2.4.10-docs/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html

Cuando los datos no son separables linealmente, SVM ofrece unas funciones kernels que permiten llevar los datos a una dimensión superior donde si sean separables (ver figura 4).

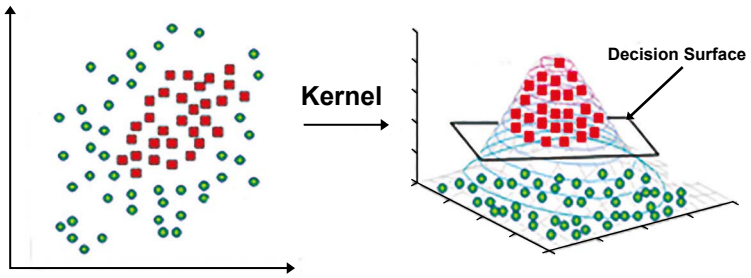


Figura 4 Función kernel.
Fuente: http://www.swarthmore.edu/NatSci/mzucker1/opencv-2.4.10-docs/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html

Árboles de decisión. Son un método de aprendizaje supervisado no paramétrico utilizado para la clasificación y la regresión. Su función es crear un modelo que prediga el valor de una variable objetivo mediante el aprendizaje de reglas simples de decisión inferidas a partir de las características de los datos. Esas reglas se representan mediante un grafo (ver figura 5) y son una serie de condiciones aplicadas a los atributos de los datos. Los nodos iniciales o intermedios del grafo representan atributos de los datos, las aristas representan las condiciones que deben cumplir para tomar ese camino y los nodos finales representan la decisión de regresión o clasificación a tomar. Para seleccionar el orden de cada uno de los atributos en el árbol, se utilizan métricas de la teoría de información, tales como, cantidad de información, entropía y ganancia de información (Giovanni, G., & Velasquez, V., 2014).

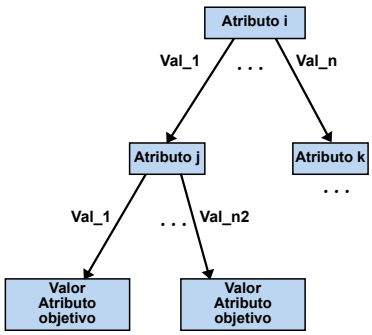


Figura 5 Árbol de decisión
Fuente: (Giovanni, G., & Velasquez, V., 2014).

Regresión lineal. Los métodos de regresión estudian la construcción de modelos para explicar o representar la dependencia entre una variable respuesta o dependiente (Y) y la(s) variable(s) explicativa(s) o independiente(s), X. Si la relación entre esos dos tipos de variables es lineal y el número de variables explicativas o independientes es de 1, la regresión es simple lineal, en el caso de que la relación sea lineal, pero haya más de una variable explicativa la regresión es lineal multivariable.

La regresión lineal simple se modela mediante la ecuación 1; donde ε es el error generado por valores aleatorios. Si ε es despreciado el modelo se reduce a la ecuación de una recta donde β0 es el corte con el eje Y y β1 es la pendiente. β0 y β1 son llamados estimadores y se calculan normalmente por el método de mínimos cuadrados con el fin de reducir el error entre los valores reales y los generados por la ecuación de la recta.

$$y = \beta_0 + \beta_1 x + \varepsilon$$

Ecuación 1 Modelo de regresión lineal simple

La precisión de las predicciones realizadas con este modelo dependerá del grado de asociación lineal existente entre las variables y la bondad del ajuste de la recta de regresión a los datos observados. Para medir este grado de linealidad y esta bondad de ajuste se utilizan dos coeficientes: el de correlación lineal de Pearson y el de determinación. En la figura 6 se pueden observar varios valores de coeficientes de correlación.

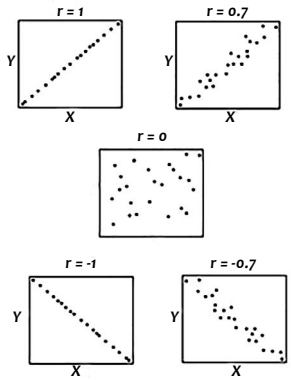


Figura 6 Factor de correlación de Pearson (r).
Fuente: Autor.

Redes Neuronales Artificiales (ANN). Como su nombre lo indica es una técnica de aprendizaje de máquina que se inspira en la estructura y el funcionamiento de las redes neuronales de nuestro cerebro. Utiliza unidades de procesamiento básicas que imitan la operación eléctrica de las neuronas mediante una función de activación que entrega una salida en función de las entradas, las cuales a su vez provienen de las salidas de otras neuronas. La conexión entre salida y entrada de las neuronas posee un factor de amplificación o de atenuación al cual se le llama peso; la actualización de estos pesos representa el proceso de aprendizaje, es decir la sinapsis (ver figura 7).

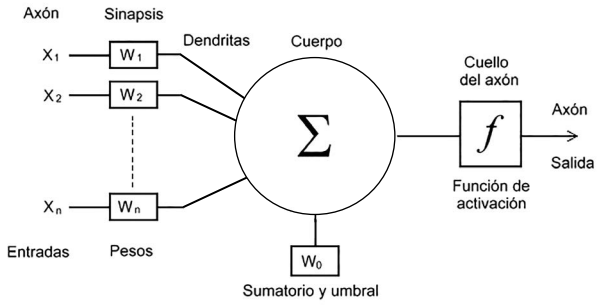


Figura 7 Neurona artificial.
Fuente: Autor

La forma como se conecten y operen las neuronas entre sí definen la arquitectura de la red. Una de las arquitecturas más utilizadas es la multi-layer perceptrón. Esta arquitectura es muy utilizada en tareas de clasificación y consiste en una capa de entrada con una cantidad de neuronas igual al número de rasgos a tener en cuenta para la clasificación, una o más capas ocultas y una capa de salida con un número de neuronas igual al número de clases, de tal forma que la neurona que se activa define el resultado de la clasificación. En la figura 8 se puede ver un ejemplo de red neuronal multi-layer perceptrón totalmente conectada con una capa de entrada de 4 neuronas, dos capas ocultas de 5 y 7 neuronas respectivamente y una capa de salida de tres neuronas que permitiría realizar una clasificación de 3 clases o categorías.

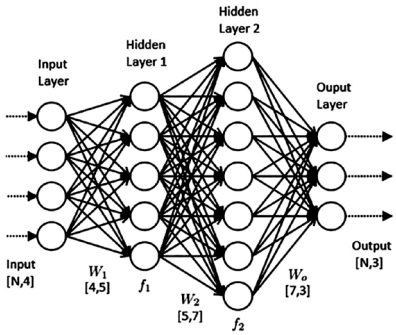


Figura 8 Red Neuronal Artificial Multi-layer Perceptrón
Fuente: https://www.datasciencecentral.com/profiles/blogs/how-to-configure-the-number-of-layers-and-nodes-in-a-neural.

¿Qué es el aprendizaje de máquina profundo?

Al inicio de esta introducción se define el aprendizaje de las máquinas como la acción de generalizar comportamientos a partir de una información suministrada en forma de ejemplos. La pregunta a resolver ahora, es ¿Qué hace que ese aprendizaje de máquina sea profundo?. El término profundo hace referencia a la capacidad de identificación y extracción automática de rasgos mediante capas consecutivas conformadas por unidades de procesamiento no lineales que permiten hacer una abstracción jerárquica de características (LeCun, Y., Bengio, Y., & Hinton, G., 2015).

Las técnicas de aprendizaje de máquina convencionales (no profundo) requieren un preprocesamiento para extraer rasgos identificados o determinados de forma no automática, es decir el aprendizaje de máquina convencional no involucra la identificación de rasgos, este es un proceso que se debe definir por humanos y que influye altamente en la precisión de la técnica. Las técnicas de aprendizaje de máquina profundo extienden el aprendizaje a la identificación y extracción de rasgos de tal forma que la máquina "aprende" cuáles son los rasgos que definen una clase de datos de entrada y cómo se deben extraer estos rasgos.

¿Qué son las redes neuronales convolucionales?

Una de las técnicas de aprendizaje profundo más utilizada y de mayor precisión son las redes neuronales convolucionales. Una red neuronal convolucional presenta una arquitectura conformada por dos bloques secuenciales, el primero cumple el propósito de detección de características o rasgos mediante la aplicación iterativa de filtros y el segundo se encarga de tomar la decisión (regularmente de clasificación) mediante una red neuronal totalmente conectada (convencional) (Krizhevsky, A., Sutskever, I., & Hinton, G.E., 2012). En la figura 9 se puede observar la arquitectura general de una red neuronal convolucional utilizada para la clasificación de imágenes.

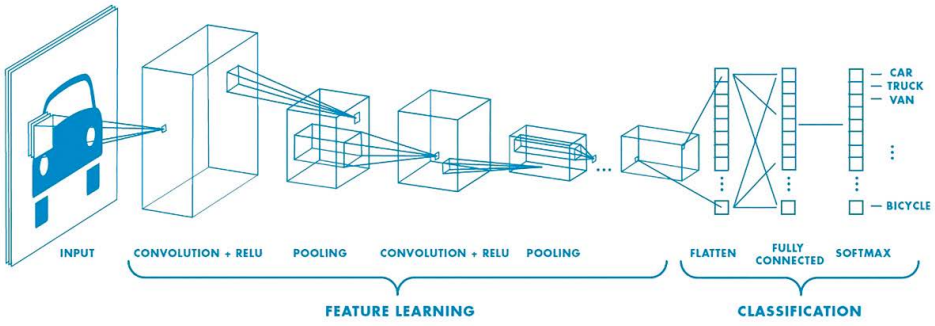


Figura 9 Arquitectura general de una red neuronal convolucional.
Fuente: https://la.mathworks.com/solutions/deep-learning/convolutional-neural-network.html

El primer bloque se encarga del aprendizaje profundo, es decir de la extracción automática de rasgos o características. Para cumplir esta tarea, este bloque posee una gran cantidad de capas conectadas entre sí de forma secuencial (la salida de una es la entrada de la siguiente); cada capa representa un filtro. Normalmente los filtros son 2 o 3 que se van aplicando iterativamente. Los filtros más usados son:

- Convolución: Esta capa/filtro aplica una máscara mediante una operación de convolución que permite resaltar rasgos o características propias de los datos de entrada.
- ReLU: La unidad lineal rectificada tiene una función similar a un

rectificador de media onda en términos de electrónica, es decir lleva a cero los valores negativos y mantiene los valores positivos. Esto se realiza con el fin de conseguir un entrena miento más rápido y efectivo. Esta rectificación se conoce como activación, ya que solo las características activadas prosiguen su camino hacia la siguiente capa.

- Pooling: Esta capa realiza una tarea de agrupación con el fin de simplificar la salida mediante la disminución no lineal de la tasa de muestreo, lo que reduce el número de parámetros que necesita la red para aprender.

Las técnicas de aprendizaje de máquina convencionales (no profundo) requieren un preprocesamiento para extraer rasgos identificados o determinados de forma no automática, es decir el aprendizaje de máquina convencional no involucra la identificación de rasgos, este es un proceso que se debe definir por humanos y que influye altamente en la precisión de la técnica. Las técnicas de aprendizaje de máquina profundo extienden el aprendizaje a la identificación y extracción de rasgos de tal forma que la máquina "aprende" cuáles son los rasgos que definen una clase de datos de entrada y cómo se deben extraer estos rasgos.

¿Qué es la computación heterogénea CPU-GPU?

A través de la historia de la computación, el paradigma de desarrollo y evolución de los procesadores se había enfocado en el aumento de su capacidad de cómputo mediante el incremento de la frecuencia de reloj, con el objeto de ejecutar una mayor cantidad de instrucciones en el menor tiempo posible. Sin embargo, desde 2003 debido al consumo de energía y los problemas de disipación de calor que limitan la construcción de procesadores que aumenten la frecuencia de reloj y el nivel de actividades productivas que puede ejecutarse en cada periodo de reloj en un único procesador, se cambió el enfoque integrando múltiples unidades de procesamiento en un mismo chip para aumentar el poder de procesamiento (de Antonio & Marina, 2005). Gracias al desarrollo de estos procesadores se abrió la posibilidad de resolver problemas computacionales que antes

hubieran sido imposibles (Alba, 2005). Estos problemas deben ser solucionados de una manera distinta a como se resuelven linealmente, tomando un problema cualquiera se divide en un conjunto de sub-problemas para resolver éstos simultáneamente sobre diferentes unidades de procesamiento.

De acuerdo a lo expuesto en el párrafo anterior, en la actualidad el desarrollo de sistemas de procesamiento se ha enfocado en producir dispositivos con la capacidad de ejecución simultánea de dos manera diferentes: La primera opción es el diseño de CPUs multi-core, optimizadas para reducir el tiempo de ejecución de procesos secuenciales (lactency cores); la segunda opción, es el diseño de sistemas de procesamiento many-thread, como por ejemplo las GPUs (Unidades de Procesamiento Gráfico) optimizadas para mejorar el desempeño (menos tiempo y menos consumo de energía eléctrica) en la ejecución de procesos paralelizables (throughput cores) (ver figura 10). Debido a que la mayoría de problemas computacionalmente intensivos poseen procesos tanto secuenciales como paralelizables, en los últimos años se ha iniciado el proceso de integración de los sistemas multi-core y los sistemas many-thread en plataformas computacionales denominadas heterogéneas (Kirk & Wen-mei, 2012).

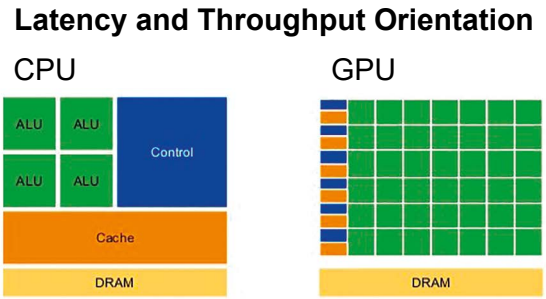


Figura 10 CPU (lactency cores) vs GPU (throughput cores).
Fuente: https://gigazine.net/gsc_news/en/20130725-40-year-cpu-history.

Una plataforma de computación heterogénea se define como un sistema conformada por lo menos de dos tipos diferentes de procesadores, normalmente, con el objeto de incorporar capacidades de procesado especializadas para realizar tareas particulares (Amar Shan, 2006). Un siste-

ma heterogéneo se conforma habitualmente por una o más CPU(s) que cumple(n) la función de unidad de procesamiento principal (llamado generalmente Host) y uno o más dispositivos de procesamiento diferentes, como por ejemplo GPUs (Graphics Processing Units), DSPs (Digital Signal Processors), FPGAs (Field Programmable Gate Arrays), que cumple(n) la función de aceleradores (ver figura 11). También se puede encontrar la integración de dos o más tipos de procesadores en un solo Chip, por ejemplo, un APU (accelerated processing unit) es un microprocesador que integra una CPU multinúcleo y una GPU mediante un bus de alta velocidad.

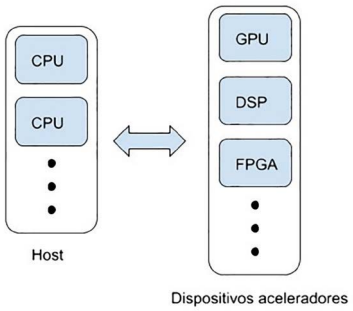


Figura 11 Plataforma heterogénea típica.
Fuente: Autor

Así como la heterogeneidad entre dispositivos de procesamiento representa una ventaja al ofrecer capacidades de procesado especializadas para realizar tareas particulares, también representa una gran desventaja desde el punto de vista del desarrollo. La heterogeneidad entre dispositivos de procesamiento se centra principalmente en la diferencia entre arquitecturas de conjuntos de instrucciones ISA (Instruction Set Architecture), por tal motivo cada uno de los tipos de dispositivos podrá contar con modelos, paradigmas y herramientas de programación totalmente diferentes, lo que conlleva a procesos de desarrollo separados con tortuosas integraciones. Los limitantes en la integración de procesos de desarrollo para los diferentes tipos de dispositivos que pueden estar involucrados en un sistema heterogéneo se han comenzado a mitigar con la creación de estándares de plataformas y modelos de programación tales como CUDA y OpenCL.

¿Cómo ayuda la computación heterogénea CPU-GPU al aprendizaje profundo?

Arriba se mencionó que el término profundo hace referencia a la capacidad de identificación y extracción automática de rasgos mediante capas consecutivas conformadas por unidades de procesamiento no lineales que permiten hacer una abstracción jerárquica de características; también se indicó que las capas más populares son Convolución, ReLU y Pooling. Aunque la complejidad de las operaciones que conforman estas capas es relativamente baja, la aplicación consecutiva e iterativa de éstas representa una tarea muy intensiva computacionalmente, debido al gran volumen de datos que debe contener el conjunto de entrenamiento para que la identificación y extracción automática de características sea altamente precisa; adicionalmente estas operaciones se aplican mediante un barrido por todos los elementos que constituyen cada dato de entrada lo que eleva sustancialmente la carga computacional.

Estas funciones que permiten la identificación y extracción automática de características, tienen una particularidad que representa una gran oportunidad para superar la intensividad computacional: las operaciones que se aplican en el barrido son totalmente independientes de bloque a bloque lo que habilita la paralelización masiva de su implementación y ejecución. Funciones tales como la Convolución, ReLU y Pooling se conforman de operaciones de complejidad media o baja que se deben aplicar (interdependientemente) muchísimas veces, esto hace que este tipo de funciones sean totalmente adecuadas para acelerarlas mediante plataformas many-thread como por ejemplo las GPU.

¿Qué pretende este libro y cómo está organizado?

El propósito de este libro es presentar una guía práctica muy simple de seguir para la implementación, entrenamiento y validación de redes neuronales convolucionales usando Keras y acelerando con GPU. La guía se desarrolla mediante un caso de estudio típico enmarcado en las clasificaciones de imágenes satelitales.

La estructura del libro corresponde a cada una de las fases que comprenden el desarrollo de un proyecto típico de clasificación de imágenes utilizando redes neuronales convolucionales: el capítulo 1 describe el caso de estudio y presenta la conformación del conjunto de datos con el cual se va a trabajar; el capítulo 2 explica las dependencias necesarias para desarrollar el proyecto y la configuración de la sesión de la GPU; el capítulo 3 trata el preprocesamiento de las imágenes que conforman el conjunto de datos, incluyendo la generación y la carga de estas imágenes; el capítulo 4 corresponde al núcleo de este libro, allí se diseña, compila y entrena el modelo de red neuronal convolucional; el capítulo 5 presenta la validación del modelo presentando y analizando los resultados de la evaluación mediante gráficas como la ROC y métricas tales como F1-score, recall, accuracy, coeficiente de kappa, etc.; el capítulo 6 presenta y analiza los resultados de la comparación de desempeño computacional del proceso de entrenamiento bajo los dos tipos de plataforma: CPU vs GPU, no solamente del modelo implementado sino también de otros modelos típicos; finalmente se exponen las conclusiones obtenidas del proceso desarrollado a través de todo el libro.

Capítulo 1

Caso de Estudio

En este capítulo se pretende exponer el caso de estudio que se desarrollará a lo largo de este libro. Además de esto, se presentará el conjunto de datos que se utilizará para llevar a cabo el desarrollo del mismo. Lo anterior, con el propósito de contextualizar al lector, dentro del problema que se trabajará.

1.1 Descripción

1.1.1 ¿Qué es una imagen?

Una imagen puede ser definida matemáticamente como una función bidimensional, $f(x, y)$, donde x y y son coordenadas espaciales (en un plano), y f en cualquier par de coordenadas es la intensidad o nivel de gris de la imagen en esa coordenada.

Cuando x, y, y los valores de f son todas cantidades finitas, discretas, se dice que la imagen es una imagen digital. Una imagen digital se compone de un número finito de elementos, cada uno con un lugar y valor específicos. Estos elementos son llamados pels, o píxelse (González y Woods,1996).

a) Una imagen continua donde se describe de forma aproximada por una serie de muestras igualmente espaciadas organizadas en forma de una matriz N x M como se indica en la ecuación 2, donde cada elemento de la matriz es una cantidad discreta y el término de la derecha representa lo que comúnmente se denomina una imagen digital:

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & . & . & . & f(0,N-1) \\ F(1,0) & f(1,1) & . & . & . & f(1,N-1) \\ . & & & & & . \\ . & & & & & . \\ . & & & & & . \\ f(M-1,0) & f(M-1,1) & . & . & . & f(M-1,N-1) \end{bmatrix}$$

Ecuación 2 Matriz N x M de cantidades discretas.

b) Otra manera de representar la imagen digital es con una notación de matrices más tradicional (ecuación 3) donde cada uno de sus elementos es un píxel o elemento de la imagen.

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & . & . & . & a_{0,N-1} \\ a_{1,0} & a_{1,1} & . & . & . & a_{1,N-1} \\ . & & & & & . \\ . & & & & & . \\ . & & & & & . \\ a_{M-1,0} & a_{M-1,1} & . & . & . & a_{M-1,N-1} \end{bmatrix}$$

Ecuación 3 Matriz de píxelse.

En cualquiera de los dos casos, no se requiere un valor especial de M y N, salvo que sean enteros positivos. En el caso del número de niveles de gris, éste es usualmente una potencia entera de 2 (ecuación 4):

$$L=2^K, K \in Z$$

Ecuación 4 Niveles de gris.

1.1.2 ¿Qué es una imagen aérea?

La fotografía aérea se obtiene por la realización de un vuelo fotogramétrico, es decir, un vuelo en el que un aeroplano sobrevuela una zona tomando repetidas fotos para componer toda la superficie. Dicha fotografía es la representación cónica de la realidad y por lo tanto está afectada por las limitaciones debidas a la perspectiva, a las que hay que sumar las deformaciones del relieve del terreno (objetos de las mismas dimensiones reales al estar más próximos al objetivo aparecerán de mayor tamaño, y viceversa), la falta de verticalidad de la toma fotográfica (objetos de considerable altura como edificios y árboles aparecerán abatidos) y las distorsiones propias del objetivo de la cámara empleada (ASPRS, 1980).

1.2 Conjunto de datos

Uno de los principales aspectos a considerar en la clasificación de imágenes es el conjunto de datos de entrenamiento que se va a utilizar. En la clasificación de imágenes aéreas se cuenta con un número limitado de conjuntos de datos públicos, uno de ellos es NWPU-RESIS45, el cual es un punto de referencia en la clasificación de imágenes de detección remota. Este conjunto de datos, fue creado por la Universidad Politécnica del Noroeste ubicada en China. Además, cuenta con un total 31.500 imágenes, distribuidas en un total de 45 clases, cada una con alrededor de 700 imágenes. Las clases de este conjunto de datos son: avión, aeropuerto, diamante de béisbol, cancha de baloncesto, playa, puente, chaparral, iglesia, tierras de cultivo circulares, nube, área comercial, residencial denso, desierto, bosque, autopista, campo de golf, campo de tierra, puerto, área industrial, intersección, isla, lago, prado, residencial medio, parque de casas móviles, montaña, paso elevado, palacio, estacionamiento, ferrocarril, estación de ferrocarril, tierras de cultivo rectangulares, río, rotonda, pista de aterrizaje, mar, barco, snowberg, residencial escaso, estadio, tanque de almacenamiento, cancha de tenis, terraza, central térmica y humedal (Cheng, G., Han, J., & Lu, X, 2017). La figura 12, presenta una muestra de las imágenes consolidadas en este conjunto de datos.

Figura 12 Muestra conjunto de datos NWPU-RESIS45.
Fuente: Autor

Otro conjunto de datos disponible y abierto al público es UC Merced Land Use, compuesto por un total de 21 clases, cada una de ellas con 100 imágenes. Las clases de este conjunto de datos son las siguientes: Agricultura, avión, diamante de béisbol, playa, edificios, chaparral, residencial denso, bosque, autopista, campo de golf, puerto, intersección, residencial medio, parque de casas móviles, puente, estacionamiento, río, pista de aterrizaje, residencial escaso, tanque de almacenamiento y cancha de tenis (Yang, Y., & Newsam, S., 2010). A continuación, se expone una muestra de este dataset (ver figura 13).

Figura 13 Muestra conjunto de datos UC Merced Land Use.
Fuente: Autor.

Con base en lo anterior,los autores de este libro realizaron un proceso de unión entre los dos conjuntos de datos presentados previamente. Sin embargo, con el propósito de simplificar el desarrollo del problema a lo largo del libro, se seleccionaron tan solo tres clases: Avión, barco y estadio. Por lo general, los conjuntos de datos utilizados para llevar a cabo clasifica-

ción de imágenes mediante redes neuronales por convolución, presentan en su distribución, imágenes de entrenamiento, imágenes de validación e imágenes de prueba.

En consecuencia, la figura 14, presenta la distribución del conjunto de datos que se utilizará a lo largo de este libro, donde un total de 1540 imágenes componen el entrenamiento, 360 imágenes son de validación y 300 imágenes son utilizadas para probar el modelo.

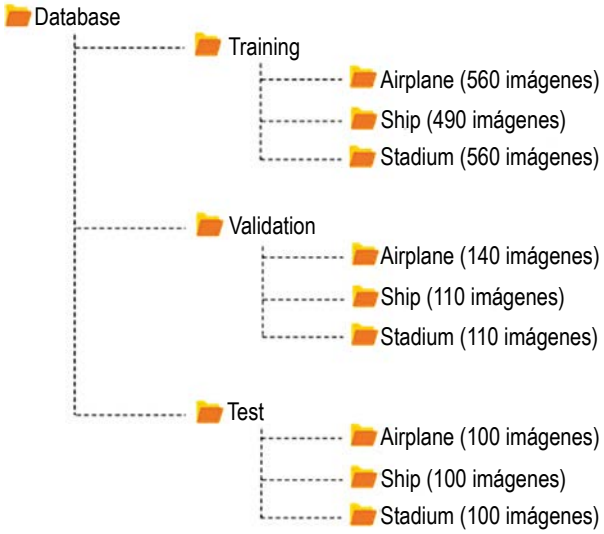


Figura 14 Distribución conjunto de datos.
Fuente: Autor.

El conjunto de datos se encuentra disponible para su descarga en la siguiente dirección.
https://drive.google.com/drive/folders/1uF0331HDofMrdL1zIxNoepohcW6_X8rY?usp=sharing.

Una vez, se ha realizado la socialización del caso de estudio a desarrollar y el conjunto de datos a utilizar en este proceso, se procede a presentar las dependencias y configuración del entorno de trabajo.

Capítulo 2

Dependencias y configuración

En este capítulo, se tiene como objetivo realizar un vistazo de cada una de las dependencias o librerías que se utilizarán a lo largo del desarrollo de este libro. Así mismo, se presenta la configuración requerida para ejectuar el código fuente sobre una Unidad de Procesamiento Gráfica (GPU).

2.1 DependenciasKeras

2.1.2 Keras

Figura 15 Logotipo de Keras.
Fuente: https://keras.io/

Keras (Chollet, F., 2015) es una API de alto nivel para desarrollar redes neuronales, está escrita en Python y se puede ejecutar sobre TensorFlow, CNTK o Theano. Permite el desarrollo fácil y rápido de dos tipos de redes neuronales de aprendizaje profundo: redes neuronales convolucionales y

redes neuronales recurrentes, también permite la combinación de estos dos tipos de redes. Para facilitar el proceso de entrenamiento, Keras se ejecuta tanto en CPU como en GPU.

2.1.2 TensorFlow

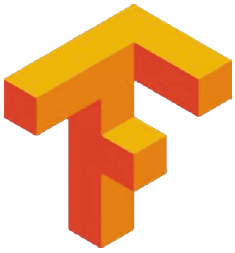


Figura 16 Logotipo de TensorFlow.
Fuente: https://www.tensorflow.org/

TensorFlow (Abadi, M., et.al., 2016) es una plataforma de código abierto que permite el desarrollo de aplicaciones de aprendizaje de máquina. Está constituida por un ecosistema integral de herramientas, librerías y recursos dirigido tanto al investigador que desea aportar al estado del arte del aprendizaje de máquina como al desarrollador que requiere crear y desplegar fácilmente aplicaciones de aprendizaje de máquina. TensorFlow fue desarrollado por Google en el marco del proyecto Google Brain y fue liberado como software de código abierto el 9 de noviembre de 2015.

2.1.3 Scikit-Learn

Figura 17 Logotipo de Scikit-Learn.
Fuente: https://scikit-learn.org

Scikit-Learn (Pedregosa, F., 2011) es un paquete de Python que incluye herramientas eficientes y simples para analizar datos mediante algoritmos de aprendizaje de máquina en tareas de regresión, clasificación y agrupación. Dentro de los algoritmos que implementa están: máquinas de vecto-

res de soporte, bosques aleatorios, Gradient boosting, K-means, DBSCAN, redes neuronales, etc.

2.1.4 Scipy

Figura 18 Logotipo de SciPy.
Fuente: https://www.scipy.org/

Scipy (Jones, E., Oliphant, T., & Peterson, P., 2001) es un ecosistema conformado por software de código abierto basado en Python y diseñados para las matemáticas, las ciencias y la ingeniería. El núcleo de Scipy está conformado por 6 paquetes: Numpy, Matplotlib, Ipython, Sympy, Pandas y las librerías propias de Scipy.

2.1.5 Numpy

Figura 19 Logotipo de NumPy.
Fuente: https://numpy.org/

NumPy es un paquete fundamental para desarrollar computación científica con Python. Su principal aporte se centra en una estructura de datos n-dimensional (denominada arreglo) muy eficiente tanto en el almacenamiento como en su operación (Van Der Walt, S., Colbert, S.C., & Varoquaux, G., 2011). Numpy no solo ofrece la estructura de datos sino también una gran biblioteca de funciones matemáticas de alto nivel para operar eficientemente dichos arreglos.

2.1.6 Matplotlib

Figura 20 Logotipo de Matplotlib.
Fuente: https://matplotlib.org/

Matplotlib (Hunter, J.D., 2007) es una librería para la generación de gráficos 2D en Python. Matplotlib tiene la capacidad de generar gráficos en ambientes estáticos o dinámicos, con datos provenientes de diferentes estructuras de datos, como por ejemplo listas, arreglos, dataframes, entre otros. Los gráficos se pueden generar desde scripts de Python, consolas de Python o Ipython, Notebooks de Jupiter o desde servidores web.

2.1.7 Procedimiento para importar dependencias

Con el propósito de ejecutar exitosamente el código de Python presentado a lo largo del desarrollo de este libro, es fundamental importar cada una de las dependencias necesarias para llevar este proceso a cabo. El siguiente fragmento de código presenta la manera de importar cada una de las librerías.

```
1.  import keras
2.  from keras.models import Sequential, load_model
3.  from keras.layers import Conv2D, MaxPooling2D,Dense, Dropout,
      Flatten, Activation
4.  from keras.callbacks import EarlyStopping, ModelCheckpoint
5.  import tensorflow as tf
6.  from keras import backend as K
7.  from tensorflow.python.client import device_lib
8.  from keras.preprocessing.image import ImageDataGenerator
9.  from keras.utils import to_categorical
10. import numpy as np
11. from scipy import interp
12. from sklearn.metrics import accuracy_score
```

```
13. from sklearn.metrics import precision_score
14. from sklearn.metrics import recall_score
15. from sklearn.metrics import f1_score
16. from sklearn.metrics import cohen_kappa_score
17. from sklearn.metrics import roc_auc_score
18. from sklearn.metrics import confusion_matrix
19. from sklearn.metrics import roc_curve, auc
20. import matplotlib.pyplot as plt
21. from itertools import cycle
```

Líneas 1-4:

```
import keras
from keras.models import Sequential, load_model
from keras.layers import Conv2D, MaxPooling2D,Dense, Dropout, Flatten,
      Activation
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

En estas líneas se cargan las librerías relacionadas con los modelos de redes neuronales por convolución, por lo tanto, están encargadas de hacer posible el diseño, compilación, entrenamiento y carga de los modelos.

Líneas 5-7:

```
import tensorflow as tf
from keras import backend as K
from tensorflow.python.client import device_lib
```

Las anteriores líneas de código permiten importar las dependencias requeridas para la definición de sesión de trabajo en la GPU y posterior ejecución del código fuente sobre dicho dispositivo.

Línea 8:

```
from keras.preprocessing.image import ImageDataGenerator
```

Dado que el conjunto de datos es un factor importante para el desarrollo de redes neuronales por convolución, se hace necesario importar el módulo de la línea 8, para poder realizar un preprocesamiento a las imágenes que componen dicho conjunto.

Líneas 9-19:

```
from keras.utils import to_categorical
import numpy as np
from scipy import interp
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import cohen_kappa_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_curve, auc
```

Estas líneas de código permiten cargar los distintos módulos de las librerías necesarias para llevar a cabo la evaluación del modelo de la red neuronal por convolución. Lo anterior, haciendo uso de distintas métricas como: Curvas de ROC, Precision Score, Accuracy Score, F1 Score, Recall Score, Coeficiente de Kappa y Matriz de Confusión.

Líneas 20-21:

```
import matplotlib.pyplot as plt
from itertools import cycle
```

Finalmente, las líneas anteriores, tienen el propósito de permitir presentar resultados gráficos durante el desarrollo de este caso de estudio.

2.2 Configuración de la sesión GPU

La interacción entre librerías como Keras y Tensorflow, generan la posibilidad de trabajar distintos procesos relacionados con la implementación de redes neuronales por convolución en una arquitectura paralelizada, haciendo uso de dispositivos como GPU. Sin embargo, si usted no cuenta con una tarjeta graficadora puede omitir el procedimiento de configuración de sesión en GPU y todo el proceso expuesto en el libro se ejecutará por defecto en CPU.

2.2.1 Procedimiento para configurar la sesión de GPU

El siguiente fragmento de código fuente, permite realizar la configuración de la sesión de trabajo en GPU para la ejecución de tareas propias de Keras.

```
22. print(device_lib.list_local_devices())
23. K.tensorflow_backend._get_available_gpus()
24. config = tf.ConfigProto()
25. sess = tf.Session(config=config)
26. keras.backend.set_session(sess)
```

Línea 22:

```
print(device_lib.list_local_devices())
```

Esta línea de código, mediante la función list_local_devices (), pretende mostrar por consola el listado de dispositivos disponibles localmente. Al ejecutar esta línea de código se puede presentar algo similar a lo que se presenta en la figura 21.

```
[name: "/device:CPU:0"
 device_type: "CPU"
 memory_limit: 268435456
 locality { }
 incarnation: 17310413457301815335,
 name: "/device:GPU:0"
 device_type: "GPU"
 memory_limit: 6686052843
 locality { bus_id: 1 links { } }
 incarnation: 3753028348053837586
 physical_device_desc: "device: 0,
 name: GeForce GTX 1070,
 pci bus id: 0000:01:00.0,
 compute capability: 6.1" ]
```

Figura 21 Dispositivos locales.
Fuente: Autor.

La figura 21, expone que, el computador utilizado tiene un dispositivo CPU y un dispositivo GPU, el cual presenta a detalle características como la marca de Unidad de procesamiento gráfico que se posee, en este caso una GeForce GTX 1070, además de la memoria límite.

Línea 23:

```
K.tensorflow_backend._get_available_gpus()
```

Una vez se han encontrado los dispositivos que se posee localmente,- se procede mediante la función _get_available_gpus() a obtener los dispositivos GPUs que se encuentra disponibles. Al ejecutar esta línea de código, se puede presentar el siguiente resultado (ver figura 22).

```
['/job:localhost/replica:0/task:0/device:GPU:0']
```

Figura 22 Dispositivos GPU disponibles.
Fuente: Autor.

Línea 24-26:

```
config = tf.ConfigProto()
sess = tf.Session(config=config)
keras.backend.set_session(sess)
```

Como primera instancia, en estas líneas, la función ConfigProto() se utiliza para configurar la sesión de Tensorflow, dado que a esta función no se pasa ningún tipo de parámetros, por defecto se inicializan todos los dispositivos GPU disponibles, en este caso tan solo se inicializará uno y dicha configuración se almacena en la variable config. Acto seguido, en la variable sess, se almacena la creación de la sesión de Tensorflow usando la función Session(), pasando como parámetros la configuración establecida anteriormente. Por último, se "setea" como sistema de fondo la sesión configurada en GPU, invocando el módulo Backend e implementando su función set_session() recibiendo como parámetro la sesión de Tensorflow.

Finalizado tanto el proceso de importación de cada una de la librerías necesarias para la ejecución del código fuente asociado al caso de estudio a desarrollar como el proceso de configuración de la sesión bajo una GPU, se prosigue en el siguiente capítulo a realizar el preprocesamiento de imágenes.

Capítulo 3

Preprocesamiento de Imágenes

En este capítulo, se dará a conocer los fundamentos del preprocesamiento de imágenes y algunos de sus tareas clave, tales como: la transformación, la generación y la carga de imágenes.

El preprocesamiento de imágenes con fines de entrenamiento de una red neuronal convolucional puede estar enrutado en dos sentidos:

- Aplicar transformaciones para favorecer o facilitar tanto el suministro como el procesamiento de las imágenes en la red neuronal, como por ejemplo cambiar tamaños, hacer escalamientos, normalizaciones, entre otras.
- Aplicar transformaciones para aumentar el conjunto de datos de entrenamiento generando nuevas imágenes a partir de la alteración de las existentes con el propósito de mejorar el resultado del entrenamiento. Las alteraciones pueden ser rotaciones, reflejos, cambio de posiciones, entre otras, y se pueden aplicar de forma aleatoria (ver figura 23).

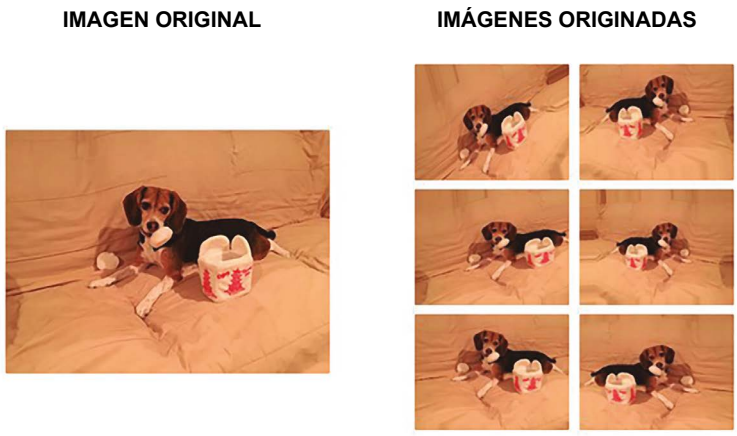


Figura 23 Aumentando el conjunto de datos de entrenamiento mediante transformaciones. Fuente Imagen modificada de https://www.pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/

3.1 Generador de Imágenes

Los dos tipos de preprocesamiento mencionados arriba se puede realizar en Keras a través de los métodos de la clase ImageDataGenerator como por ejemplo apply_transform, que mediante uno de sus parámetros permite aplicar operaciones como rotaciones, reflejos, inversión, zoom o mediante el método random_transform que permite aplicar las mis operaciones, pero de forma aleatoria.

3.1.1 Procedimiento para establecer el generador de imágenes

El siguiente fragmento de código fuente, presenta la manera de invocar el generador de imágenes, propio de la librería Keras.

```
27. ruta_dataset_entrenamiento = "ruta/local/dataset/training"
28. ruta_dataset_prueba = "ruta/local/dataset//test"
29. ruta_dataset_validacion = "ruta/local/dataset//validation"
30. train_datagen = ImageDataGenerator(rescale=1./255)
31. test_datagen = ImageDataGenerator(rescale=1./255)
32. validation_datagen = ImageDataGenerator(rescale=1./255)
```

Líneas 27-29:

```
ruta_dataset_entrenamiento = "ruta/local/dataset/training"
ruta_dataset_prueba = "ruta/local/dataset//test"
ruta_dataset_validacion = "ruta/local/dataset//validation"
```

En estas líneas se realiza una tarea simple, tanto solo se definen las rutas locales donde se encuentran alojadas las imágenes de entrenamiento, validación y prueba.

Líneas 30-32:

```
train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)
validation_datagen = ImageDataGenerator(rescale=1./255)
```

Adicionalmente, se deben inicializar los generadores de imágenes de cada sub conjunto de datos. Lo anterior, mediante la función ImageDataGenerator(), en este caso, al establecer un factor de 1/255 de rescale, se tomará cada una de las imágenes en su proceso de carga y se multiplicará píxel a píxel por dicho factor.

3.2 Carga de Imágenes

Como se ha expuesto anteriormente, el conjunto de datos es un aspecto primordial en los métodos de aprendizaje profundo, es por esto que, es indispensable tomar lel directorio fuente donde se encuentran alojadas las imágenes y cargarlas a un arreglo en memoria. Adicionalmente, dado que se emplea aprendizaje supervisado, se debe generar las etiquetas correspondientes a cada una de las imágenes cargadas.

3.2.1 Procedimiento para cargar las imágenes

El código fuente expuesto a continuación, tiene como propósito realizar la carga de cada una de las imágenes que componen los subconjuntos de datos, entendiéndose como, datos de entrenamiento, validación y prueba.

```
33. train_generator = train_datagen.flow_from_directory(ruta_dataset
      _entrenamiento,target_size=(80,80),color_mode='rgb',batch_size
      =32,class_mode='categorical',shuffle=True)
34. validation_generator = validation_datagen.flow_from_directory(
      ruta_dataset_validacion,target_size=(80,80),color_mode='rgb',b
      atch_size=32,class_mode='categorical',shuffle=True)
35. test_generator = test_datagen.flow_from_directory(ruta_dataset_
      prueba,target_size=(80,80),color_mode='rgb',batch_size=1,class
      _mode='categorical',shuffle=False)
```

Línea 33:

```
train_generator = train_datagen.flow_from_directory(ruta_dataset
      _entrenamiento,target_size=(80,80),color_mode='rgb',batch_size
      =32,class_mode='categorical',shuffle=True)
```

En esta línea se utiliza el generador de entrenamiento mediante la función datagen_flow_from_directory(), la cual sirve para cargar un conjunto de imágenes a partir de una dirección local. Adicionalmente, esta función permite dividir y agrupar las imágenes cargadas de acuerdo a su distribución en su carpeta local. Lo anterior, es fundamental en el entrenamiento de máquina supervisado, donde se hace necesario tener la referencia o identificador de cada uno de los datos de entrenamiento. El primer parámetro de esta función es la dirección donde se encuentran las imágenes. Posteriormente, se define un target_size de 80x80 píxeles, es decir, cada una de las imágenes de 256x256 se redimensionarán a 80x80. Además de esto, al establecer el parámetro color_mode como 'rgb', se indica que se tendrán imágenes a color en tres bandas (red, green, blue). El siguiente parámetro batch_size, tiene como finalidad, agrupar todas las imágenes en lotes de 32 imágenes, con sus respectivas etiquetas de identificación. Acto seguido, se define el parámetro de mode_class, en donde se estipula un modo categórico, el cual tomará importancia en la fase de entrenamiento del modelo de la red neuronal por convolución. Finalmente, al establecer como verdadero el parámetro shuffle, se genera la carga y agrupación de las imágenes en lotes, de manera aleatoria. La figura 24 presenta el resultado obtenido al ejecutar esta línea de código.

```
Found 1540 images belonging to 3 classes.
```

Figura 24 Resultado train_generator.
Fuente: Autor.

Como se ha especificado con anterioridad, el conjunto de datos de entrenamiento tiene un total de 1540 imágenes distribuidas en 3 clases.

Línea 34:

```
validation_generator = validation_datagen.flow_from_directory(
      ruta_dataset_validacion,target_size=(80,80),color_mode='rgb',b
      atch_size=32,class_mode='categorical',shuffle=True)
```

Esta línea de código, es prácticamente igual a la línea 33, tan solo con una excepción, en este caso se cargan las imágenes que se utilizarán en la validación del modelo. Al ejecutar este código, se debe presentar el siguiente resultado (ver figura 25).

```
Found 360 images belonging to 3 classes.
```

Figura 25 Resultado validation_generator.
Fuente: Autor.

En este caso, al ejecutar esta línea se encontraron 360 imágenes distribuidas en aviones, barcos y estadios.

Línea 35:

```
test_generator = test_datagen.flow_from_directory(ruta_dataset_
      prueba,target_size=(80,80),color_mode='rgb',batch_size=1,class
      _mode='categorical',shuffle=False)
```

Finalmente, se deben cargar las imágenes que se utilizarán para la etapa de evaluación del modelo. La línea 35, posee dos diferencias frente a la carga de imágenes de entrenamiento y validación. La primera de ellas es el número de imágenes por lote, en este se indica que, un lote estará conformado por tan solo una imagen. La segunda, hace referencia al orden de carga de imágenes, por lo tanto, al setear como falso

este parámetro, las imágenes se cargan en el orden en que se encuentran en las carpetas. De acuerdo con la figura 26, al ejecutar esta línea de código, se encontraron un total de 300 imágenes prueba, es decir, 100 para cada una de las clases.

```
Found 300 images belonging to 3 classes.
```

Figura 26 Resultado test_generator.
Fuente: Autor.

Después de aplicar la fase de preprocesamiento de imágenes al conjunto de datos utilizado en este libro, se da lugar a la presentación del modelo de la red neuronal por convolución empleado en esta investigación.

Capítulo 4

Modelo de la Red Neuronal por Convolución

A lo largo de este capítulo, se explicarán las fases de diseño, compilación y entrenamiento de un modelo de redes neuronales por convolución. Para llevar esto acabo, se explicarán las distintas capas de este tipo de modelos y los hiperparámetros de compilación y de entrenamiento.

4.1 Diseño del Modelo

Keras permite construir dos tipos de modelos de redes neuronales, uno secuencial y uno funcional. El secuencial es el modelo más utilizado debido a que permite implementar fácilmente la arquitectura típica multicapa de una red neuronal donde las salidas de una capa representan las entradas de la capa siguiente (Torres, J., 2018). El modelo funcional, opera como una especie de API donde el proceso de diseño del modelo es más complejo, pero permite mayor flexibilidad para permitir arquitecturas atípicas.

Un modelo secuencial de red neuronal está conformado por bloques conectados en forma de tubería donde cada bloque implica aplicar una función a los datos de entrada.Cada de uno de esos bloques es llamado capa. Cada tipo de capa representa una función diferente que se le aplica a los datos de entrada. Keras dispone de un gran número de capas, dentro de éstas se encuentran las típicas para diseñar una red neuronal convolucional: capas de convolución de 1D, 2D y 3D, capas de pooling de 1D, 2D, 3D usando máximo o promedio, capas de dropout, capas de flatten y capas tipo dense.

4.1.1 Capas de convolución

Este tipo de capa permite aplicar un número específico de filtros o kernels a los datos de entrada mediante una operación de convolución, es decir el kernel recorre todos los datos de entrada y en cada posición se obtiene un valor correspondiente a la media de los productos elemento a elemento del kernel y del bloque seleccionado de los datos (ver figura 27). El resultado de la aplicación de cada kernel tiene la misma dimensión de los datos de entrada y habrá tantos resultados como filtros aplicados, por ejemplo, en la figura 28 se puede observar la aplicación de 3 kernels a una imagen de entrada y como resultado se obtienen 3 imágenes filtradas que resaltan un patrón en función del kernel aplicado.

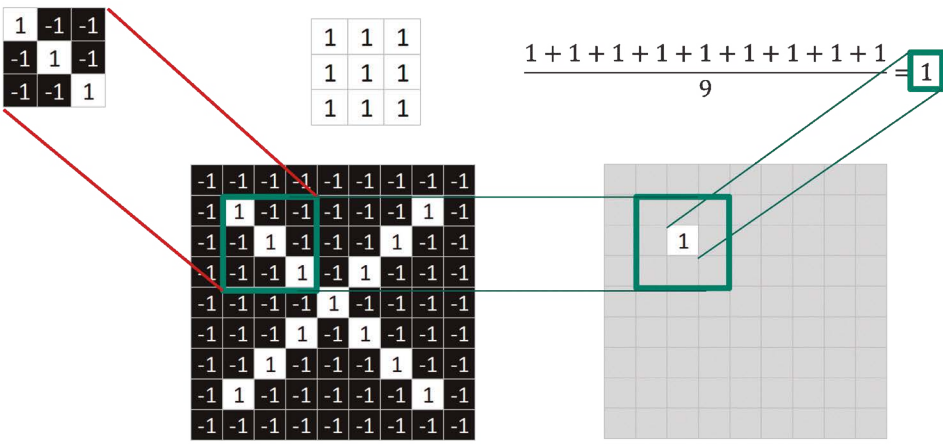


Figura 27 Operación de convolución.
Fuente: https://end-to-end-machine-learning.teachable.com

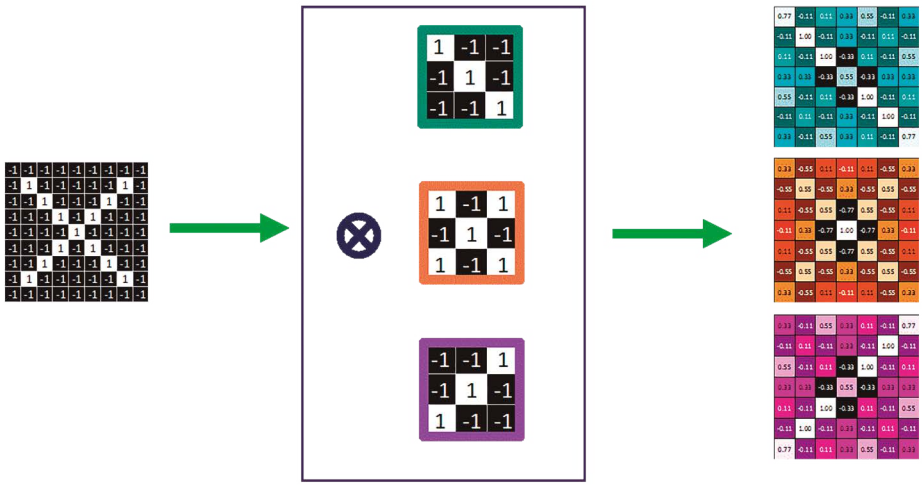


Figura 28 Resultado de la aplicación de 3 kernels.
Fuente: https://end-to-end-machine-learning.teachable.com

En el proceso de construcción de una capa de convolución en Keras se deben especificar algunos parámetros, como por ejemplo el número de filtros a aplicar, el tamaño de esos filtros, el relleno o no de los datos de entrada (si se requiere que los datos filtrados mantengan el mismo tamaño que los de entrada, se debe aumentar en 1 el tamaño de los datos de entrada en cada una de sus dimensiones, eso se hace rellenando de ceros como se puede ver en la figura 29) y la función de activación que es la operación aplicada a los datos después de la convolución (en la figura 30 se pueden ver algunas funciones de activación).

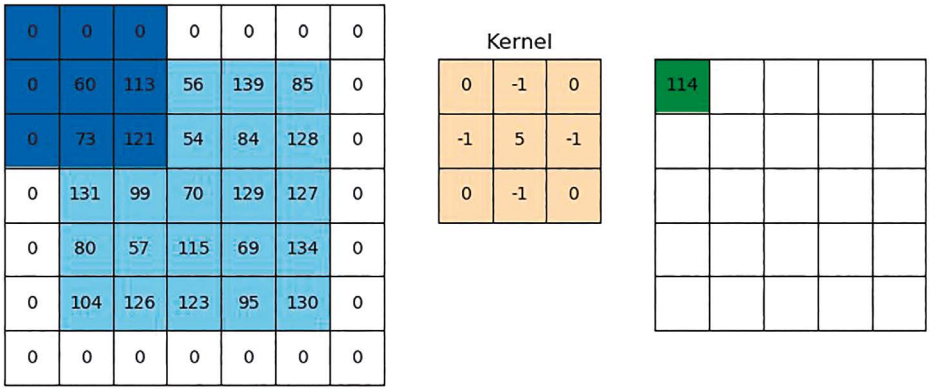


Figura 29 Relleno de la imagen de entrada.
Fuente: https://www.pyimagesearch.com/2018/12/31/keras-conv2d-and-convolutional-layers/

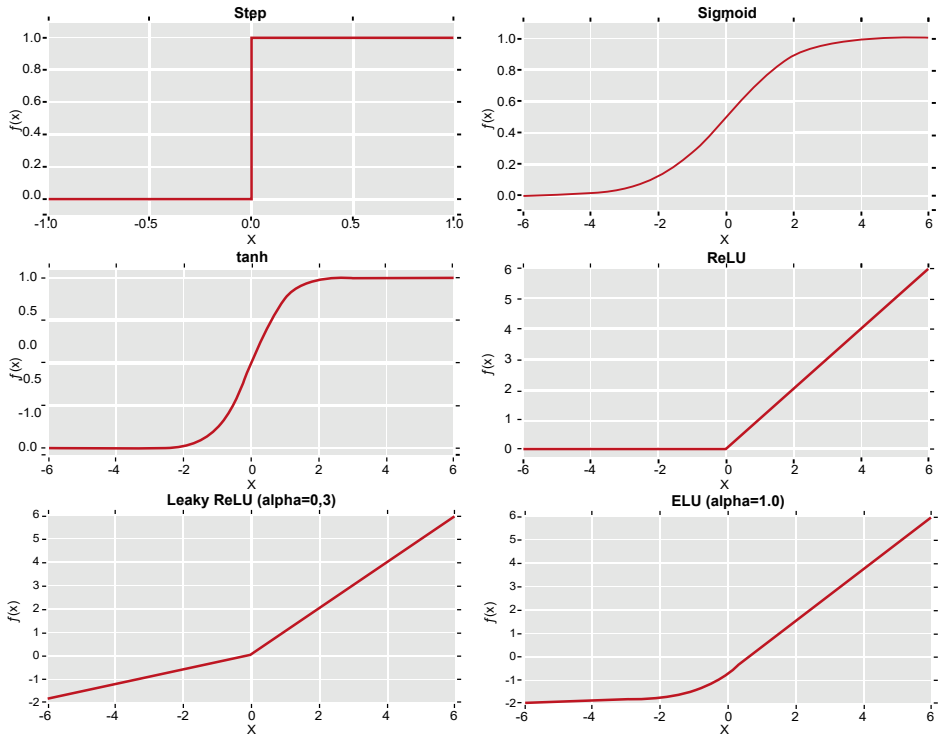


Figura 30 Funciones de activación.
Fuente: https://www.pyimagesearch.com/2018/12/31/keras-conv2d-and-convolutional-layers/

4.1.2 Capas pooling

Las capas poolíng permiten reducir el tamaño de los datos filtrados manteniendo el patrón resaltado por la operación de convolución, de esta manera se lleva a cabo una abstracción jerárquica e iterativa de rasgos. La reducción de datos se realiza mediante un agrupamiento empleando principalmente el máximo o el promedio (en la figura 31 se puede ver una operación pooling por máximo).

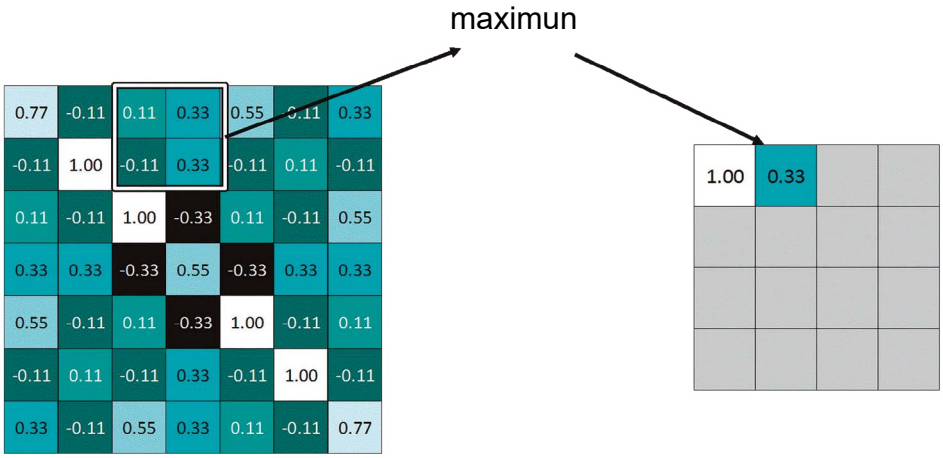


Figura 31 Operación pooling por máximo.
Fuente: https://end-to-end-machine-learning.teachable.com

4.1.3 Capas dropout

Dropout es una técnica que permite regularizar la red neuronal evitando el sobreajuste y favoreciendo la generalización mediante la desactivación aleatoria de neuronas durante el proceso de aprendizaje (Srivastava, N., et.al., 2014). La inhabilitación de algunas neuronas obliga a que otras neuronas deban intervenir para que la decisión final no se altere, esto impide que cada decisión de la red no dependa específicamente de los pesos de unas pocas neuronas. En Keras esta técnica se implementa mediante una capa que durante el entrenamiento lleva a cero una cantidad de elementos de los datos de entrada seleccionados aleatoriamente, la proporción de datos llevados a cero se pueden definir mediante un parámetro del constructor de la capa (ver figura 32).

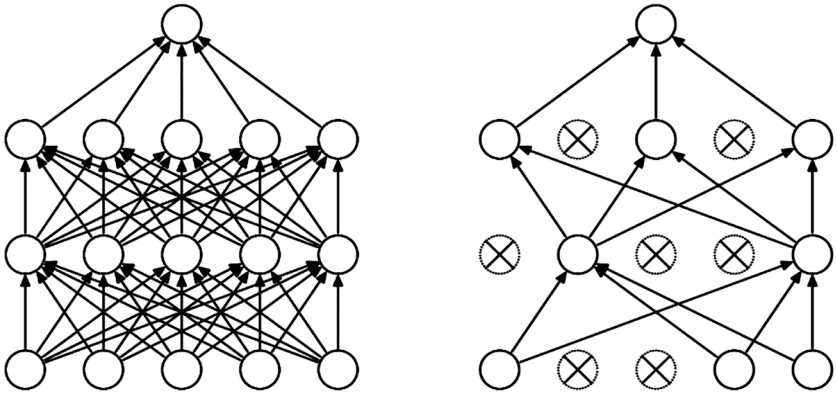


Figura 32 Dropout
Fuente (Srivastava, N., et. al., 2014).

4.1.4 Capas flatten

Estas capas toman los resultados de la sección de detección y extracción automática de rasgos y los representa en una estructura unidimensional concatenándolos todos en un único vector que se convertirá en la entrada de una red neuronal convencional totalmente conectada (ver figura 33).

Figura 33 Efecto de una capa flatten.
Fuente: https://end-to-end-machine-learning.teachable.com

4.1.5 Capas dense

Una capa dense corresponde a una capa de una red neuronal convencional totalmente conectada (ver figura 34), se utilizan en la parte final de una red neuronal convolucional, es decir en la sección donde se realiza la clasificación en función de los rasgos identificados y extraidos. Dentro de los parámetros que se configuran en este tipo de capa está el número de neuronas y la función de activación. Para el caso de un problema de clasificación multiclase, una de las funciones de activación más utilizadas es Softmax. Esta función es una generalización de una función logística que permite mapear un vector n-dimensional (de logists) a la distribución de probabilidad sobre K diferentes posibles salidas (ver figura 35).

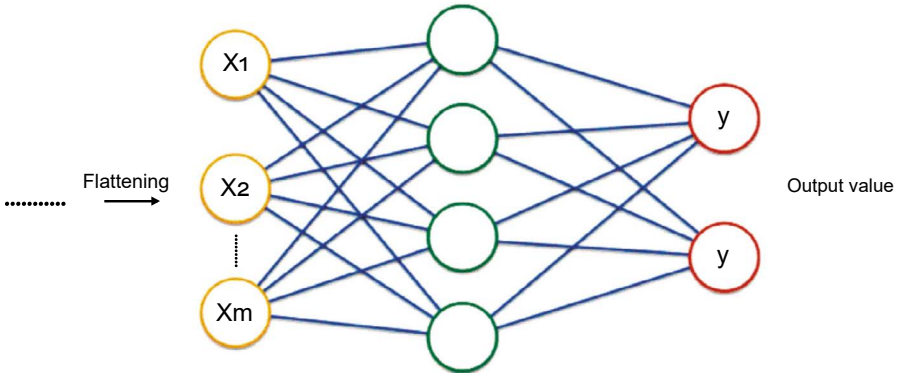


Figura 34 Ejemplo de capas dense.
Fuente: https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step--full-connection/

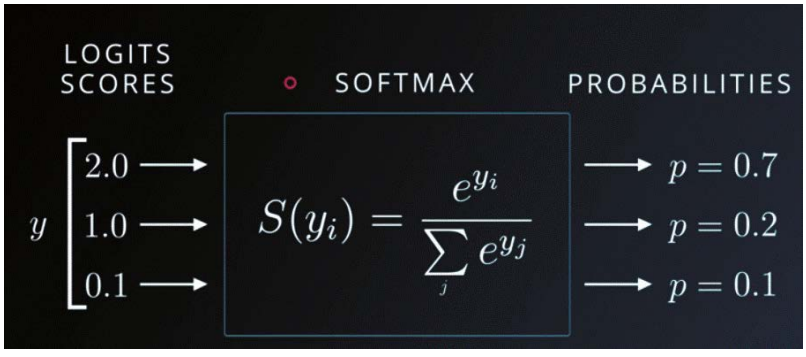


Figura 35 Función softmax.
Fuente: https://medium.com/data-science-bootcamp/understand-the-softmax-function-in-minutes-f3a59641e86d

4.1.6 Modelo para el caso de estudio

Para darle solución al caso de estudio se plantea un modelo basado en Burachonok (2017), el cual posee 4 bloques de identificación y extracción de rasgos, y un bloque de clasificación. Los bloques de identificación y extracción de rasgos están conformados por 3 capas: convolución, pooling por máximo y dropout. Todas las capas de pooling hacen un agrupamiento por ventanas de 2x2 de tal forma que el tamaño de la imagen filtrada se reduce a la mitad en cada una de sus dimensiones, de esta manera mediante 4 capas de dropout se reducen los datos de entrada de una dimensión de 80x80 elementos a 5x5 elementos. Todas las capas de dropout de este bloque se configuran para que inhabiliten el 25% de las neuronas durante el proceso de entrenamiento. En cuanto a las capas de convolución se configuran de tal forma que posean una función de activación ReLU y un tamaño de kernel de 3x3 para las 3 primeras capas y de 10x10 para la última capa. El bloque de clasificación se conforma de una capa flatten que le suministra las entradas a una capa dense de 512 neuronas y una función de activación ReLU, continúa con una capa dropout que inhabilitad la mitad de las neuronas en el proceso de entrenamiento y finalmente una capa dense de 3 neuronas y una función de activación softmax que indica cuál de las 3 categorías tiene mayor probabilidad (ver figura 36).

Figura 36 Modelo de red neuronal convolucional para el caso de estudio. Fuente: Autor.

4.1.7 Procedimiento para diseñar el modelo

El siguiente fragmento de código fuente, tiene como objetivo la creación y el diseño de la red neuronal descrita arriba para darle solución al problema de clasificación del caso de estudio.

```
36.  num_classes = 3
37.  model = Sequential()
38.  model.add(Conv2D(filters=32,kernel_size=(3,3),padding='same',
                input_shape=(80, 80, 3), activation='relu'))
39.  model.add(MaxPooling2D(pool_size=(2, 2))) #40x40
40.  model.add(Dropout(rate=0.25))
41.  model.add(Conv2D(filters=32,kernel_size=(3,3), padding='same',
                    activation='relu'))
42.  model.add(MaxPooling2D(pool_size=(2, 2))) #20x20
43.  model.add(Dropout(rate=0.25))
44.  model.add(Conv2D(filters=32,kernel_size=(3,3), padding='same',
                        activation='relu'))
45. model.add(MaxPooling2D(pool_size=(2, 2))) #10x10
46. model.add(Dropout(rate=0.25))
47. model.add(Conv2D(filters=32,kernel_size=(10,10), padding='same',
                        activation='relu'))
48. model.add(MaxPooling2D(pool_size=(2, 2))) #5x5
49. model.add(Dropout(rate=0.25))
50. model.add(Flatten())
51. model.add(Dense(units=512, activation='relu'))
52. model.add(Dropout(rate=0.5))
53. model.add(Dense(units=num_classes, activation='softmax'))
```

Línea 36:

```
num_classes = 3
```

Esta línea tan solo define la cantidad de clases finales de nuestro conjunto de imágenes, por lo tanto, ese valor hace referencia a aviones, barcos y estadio, siendo en total 3 clases.

Línea 37:

```
model = Sequential()
```

En esta línea se inicializa la variable model, como un modelo secuencial de Keras, mediante la función Sequential(). Un modelo secuencial, significa que las capas de dicho modelo se añadirán secuencialmente, es decir una tras otra y este será su orden definitivo.

Líneas 38-40:

```
model.add(Conv2D(filters=32,kernel_size=(3,3),padding='same',
                 input_shape=(80, 80, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2))) #40x40
model.add(Dropout(rate=0.25))
```

En los modelos de redes neuronales por convolución, se deben establecer bloques de extracción de características y es precisamente en estas líneas que se realiza esta tarea. Para añadir cualquier tipo de capa en un modelo secuencial se utiliza la función add(), la cual recibe por parámetros la capa que se desea agregar.

Como primera instancia, se añade una capa de convolución, donde se específica que se aplicaran un total de 32 filtros mediante el parámetro filters, los cuales se aplicaran mediante una ventana de 3x3 píxeles, es decir esta ventana recorrerá píxel a píxel cada imagen aplicando los filtros. El parámetro padding se configura en 'same' para añadir un borde de ceros a la imagen de entrada con el propósito de que la imagen resultante de la convolución tenga el mismo tamaño que la de entrada. Por lo general, en la primera capa que se agrega, se debe establecer la dimensión de entrada del modelo, en este caso mediante el parámetro input_shape se estipula una dimensión de 80x80 píxeles, con tres canales de color (R, G, B). Finalizando la capa de convolución se define la función ReLU como función de activación.

Habitualmente, después de agregar una capa de convolución, se adiciona una capa de pooling, con el fin de reducir las características y mejorar este proceso de extracción. Para efectos de este modelo, se añade una capa de max pooling, mediante la función Maxpooling2D(), dado que, se contemplan las imágenes como su representación matricial en 2 dimensiones. El parámetro pool_size, el cual se define como 2x2 píxeles, es la ventana de reducción que se aplicará.

Finalmente, para cerrar este bloque de extracción de característica, se agrega una capa de regulación mediante la función Dropout(). Para esta capa, se dispone una tasa de desconexión de neuronas del 25%. Lo anterior, al inicializar el parámetro rate en un 0.25.

Líneas 41-46:

```
model.add(Conv2D(filters=32,kernel_size=(3,3), padding='same',
                 activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2))) #20x20
model.add(Dropout(rate=0.25))
model.add(Conv2D(filters=32,kernel_size=(3,3), padding='same',
                         activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2))) #10x10
model.add(Dropout(rate=0.25))
```

En las líneas 41 a la 46, se agregan otros dos bloques de extracción de características a nuestro modelo de red neuronal por convolución. Sin embargo, se presenta un cambio, y se puede observar en la primera capa de convolución agregada, como ya se ha determinado la dimensión entrada, no es necesario volver a definirla en estas capas de convolución. Se puede observar que, estos bloques siguen exactamente el mismo patrón y parámetros que se presentan en el bloque agregado anteriormente.

Líneas 47-49:

```
model.add(Conv2D(filters=32,kernel_size=(10,10), padding='same',
                        activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2))) #5x5
model.add(Dropout(rate=0.25))
```

Estas líneas de código permiten agregar nuestro último bloque de extracción de características. No obstante, en la capa de convolución se presenta una variación frente a los bloques anteriores. Este cambio consiste en pasar de un tamaño de kernel de 2x2 píxeles a 10x10 píxeles, es decir se agrandó el tamaño de la ventana que recorre la imagen píxel a píxel aplicando los filtros. Es importe mencionar que cuando es necesario extraer las características de patrones en imágenes complejas, se recomienda utilizar varios de estos bloques.

Líneas 50-53:

```
model.add(Flatten())
model.add(Dense(units=512, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(units=num_classes, activation='softmax'))
```

En la parte final de los modelos de redes neuronales por convolución y posterior a los bloques de extracción de características, se debe presentar lo que se denomina como la capa totalmente conectada, la cual es la capa clasificadora del modelo. Como primera instancia de estas líneas de código, se agrega una capa mediante la función Flatten() para "aplanar" los datos que se han obtenido con los bloques de extracción de características, es decir que estos datos se convierten a un arreglo unidimensional. Una vez se realiza este proceso, se añade una capa que nos permite tener un número determinado de neuronas ocultas. Lo anterior, se logra mediante la función Dense(), especificando un total de 512 neuronas en el parámetro units. Posteriormente, se agrega otra capa de Dropout con una tasa de desconexión aleatoria entre neuronas del 50%. Finalmente, todos los modelos deben finalizar con una capa de neuronas ocultas, donde ese número de neuronas está determinado por número de clases que se tiene, como se puede observar

en la última línea, el parámetro units de la capa Dense(), está definido por la variable num_classes, la cual almacena el número de clases de nuestro conjunto de datos. Sin embargo, a lo largo de las capas agregadas se utilizaba la misma función de activación manteniendo homogeneidad en este aspecto, pero en este caso se utiliza la función de activación Softmax, dado que, de acuerdo con (Restrepo Rodríguez, A., et al., 2018), esta función es la más utilizada en la capa de salida y permite realizar una representación de la distribución categórica necesaria para generar la clasificación.

4.2 Compilación del Modelo

Finalizado el proceso de construcción del modelo donde se definió su arquitectura y configuración capa a capa, se continúa con la fase de compilación. La compilación de un modelo de red neuronal convolucional en keras consiste en configurar el proceso de entrenamiento mediante la definición de una función de pérdida, un optimizador y unas métricas.

4.2.1 Función de pérdida

La función de pérdida define la forma como la red puede medir su rendimiento en el proceso de entrenamiento con el fin de poder dirigir éste en la mejor dirección. En otras palabras, la función de pérdida permite medir "que tan lejos" se encuentra una predicción del objetivo real.

Keras cuenta con las siguientes funciones de pérdida: mean_squared_error, mean_absolute_error, mean_absolute_percentage_error, mean_squared_logarithmic_error, squared_hinge, hinge, categorical_hinge, logcosh, categorical_crossentropy, sparse_categorical_crossentropy, binary_crossentropy, kullback_leibler_divergence, poisson, cosine_proximity, is_categorical_crossentropy.

En los problemas similares al caso de estudio de este libro, es decir problemas de clasificación multiclase con etiqueta sencilla la función de pérdida siempre debería ser la entropía categórica cruzada (categorical_crossentropy) ya que ésta minimiza la distancia entre las distribuciones de probabilidad producidas por la red y la verdadera distribución de los objetivos.

4.2.2 Optimizador

El optimizador es el mecanismo a través del cual la red se actualiza basado en la función de pérdida. Específicamente el optimizador actualiza los parámetros de la red con el fin de minimizar (o en algunos casos maximizar) la función de pérdida. Los algoritmos de optimización se pueden agrupar en dos categorías:

• Los de primer orden, donde se encuentran todos aquellos algoritmos que minimizan o maximizan la función de pérdida usando sus valores gradientes con respecto a los parámetros de la red. La derivada de primer orden informa si la función incrementa o decrementa en un punto específico (la derivada es una línea recta tangente a la función en el punto determinado). El algoritmo más usado en esta categoría es el gradiente descendente.

• Los de segundo orden, donde se encuentran todos los algoritmos que utilizan la derivada de segundo orden (Hessian) para minimizar o maximizar la función de pérdida. La segunda derivada informa sobre la curvatura de la función. Hessian es una matriz conformada por las derivadas parciales de segundo orden.

Keras cuenta con las siguientes funciones de optimización: SGD, RMSprop, Adagrad, Adadelta, Adam, Adamax, Nadam.

4.2.3 Métricas

En Keras una métrica es una función que permite medir el desempeño del modelo; se comporta de forma similar a las funciones de pérdida con la diferencia que las métricas se emplean para evaluar, pero no para ajusta el proceso de entrenamiento. Keras cuenta con las siguientes métricas: accuracy, binary_accuracy, categorical_accuracy, sparse_categorical_accuracy, top_k_categorical_accuracy, sparse_top_k_categorical_accuracy, cosine_proximity, también se cuenta con la opción de clonar métricas o de definir sus propias métricas.

4.2.4 Procedimiento de compilación del modelo

La siguiente línea de código se utiliza para llevar a cabo la compilación del modelo creado anteriormente.

```
54. model.compile(loss='categorical_crossentropy',optimizer='adam',
      metrics=['accuracy'])
```

En esta línea se hace uso de la función compile(), propia de los modelos de Keras. Los parámetros de esta función son los siguiente:

- ***loss***, permite definir cuál función de pérdida se utilizará en fase de entrenamiento y de prueba. En este caso se implementa entropía categórica cruzada, mediante la palabra reservada ***'categorical_crossentropy'***. Es acá donde se debe referenciar el proceso de carga de imágenes, donde se estableció un modo de clase categórico, precisamente para utilizar este tipo de función de pérdida.
- ***optimizer***, hace referencial optimizador utilizado para evaluar cuanto y como corregir el peso a lo largo del aprendizaje, es decir el ritmo de aprendizaje. En este caso se utiliza el optimizador Adam.
- ***metrics***, como su nombre en inglés lo indica, son las métricas que se utilizarán para monitorear el proceso de entrenamiento, sin embargo no presenta ningún tipo de incidencia en este.

4.3 Entrenamiento del Modelo

En el aprendizaje de máquina supervisado, el entrenamiento de una red neuronal consiste en la actualización iterativa de sus pesos con el fin de que la salida predicha sea similar a la salida esperada definida por un conjunto de datos de entrenamiento. El entrenamiento de una red neuronal se realiza por épocas (epoch), que consisten en ciclos en los cuales se utiliza por completo el conjunto de datos de entrenamiento. Se debe aclarar que, aunque en cada época el conjunto de datos de entrenamiento se utiliza por completo la forma como se suministra a la red neuronal es particionado en bloques (batch), de tal forma que al final de cada bloque se actualizan los parámetros de la red teniendo en cuenta la función de perdida y el optimizador definido durante la compilación del modelo.

Keras dispone de un conjunto de funciones denominadas callback que se aplican en un estado determinado de entrenamiento con el fin de monitorearlo o de intervenirlo para mejorar su eficiencia y/o calidad. Dentro de las funciones callback más utilizadas están:

- Detección temprana (EarlyStopping) que como su nombre lo indica detiene el entrenamiento cuando un parámetro monitoreado haya dejado de mejorar, evitando así el sobreentrenamiento.
- Punto de chequeo (ModelCheckpoint), que guarda el modelo después de cada época.
- Reducción de tasa (ReduceLROnPlateau), que reduce la tasa de aprendizaje cuando una métrica ha dejado de mejorar.
- Registro a CSV (CSVLogger), que envía a un archivo CSV los resultados de cada época.
- Monitoreo remoto (RemoteMonitor), que envía toda la información de los eventos del entrenamiento a un servidor.

4.3.1 Procedimiento para entrenar el modelo

A continuación, se presenta el código fuente para entrenar el modelo de la red neuronal por convolución compilado anteriormente, además de presentar un par de gráficas asociadas a este proceso.

```
55. early_stop = EarlyStopping(monitor='val_loss', mode='min',
      verbose=1, patience=5)
56. check_point= ModelCheckpoint(filepath="ruta/local/model.h5",
      monitor='val_loss', save_best_only=True, verbose=1,mode='min')
57. callbacks = [early_stop, check_point]
58. step_size_train=train_generator.n/train_generator.batch_size
59. step_size_validation=validation_generator.n/validation_
      generator.batch_size
60. history=model.fit_generator(generator=train_generator,steps_
      per_epoch= step_size_train,validation_data = validation_
      generator,validation_steps= step_size_validation,epochs=32,
      callbacks=callbacks)
61. plt.plot(history.history['acc'])
62. plt.plot(history.history['val_acc'])
```

```
63. plt.title('model accuracy')
64. plt.ylabel('accuracy')
65. plt.xlabel('epoch')
66. plt.legend(['train', 'test'], loc='upper left')
67. plt.show()
68. plt.plot(history.history['loss'])
69. plt.plot(history.history['val_loss'])
70. plt.title('model loss')
71. plt.ylabel('loss')
72. plt.xlabel('epoch')
73. plt.legend(['train', 'test'], loc='upper left')
74. plt.show()
```

Líneas 55-57:

```
early_stop = EarlyStopping(monitor='val_loss', mode='min',verbose=1,
      patience=5)
check_point= ModelCheckpoint(filepath="ruta/local/model.h5",
      monitor='val_loss', save_best_only=True, verbose=1,mode='min')
callbacks = [early_stop, check_point]
```

En estas líneas de código se realizar la declaración de las funciones callback y se consolidan en una lista para ser invocada durante el entrenamiento del modelo. Primero, se define la variable early_stop, siendo igualada a la función EarlyStopping(), la cual como su nombre lo indica, genera una detención temprana en el entrenamiento del modelo, de acuerdo a la función de pérdida o las métricas establecidas. Sus argumentos son los siguiente:

- ***monitor***, hace referencia a la variable que se va a monitorear y de acuerdo a esa variable se lanzará la acción de detención, en este caso se monitorea el valor de la función de pérdida obtenido iteración a iteración en el proceso de validación, mediante el conjunto de datos destinado para este proceso.

- ***mode***, debe tomar el valor de auto, max o min. Cuando se define como 'min' el entrenamiento de detendrá cuando la variable mo nitoreada haya dejado de disminuir.
- ***verbose***, se utiliza tan solo para definir si los resultados de esta función se observaran en pantalla durante el entrenamiento o no. Al definirlo en 1, esto indica que si se presentaran en la consola.
- ***patience***, indica el número de iteraciones de entrenamiento que el algoritmo esperara sin tener una mejora en la variable de detención. Una vez se cumplan esta cantidad de iteraciones y no se presente mejora, se accionará la función de detección temprana. En este caso, el número de interacciones establecido es 5.

Por otro lado, la función ModelCheckpoint(), se utilizará para guardar el modelo que se va entrenando iteración tras iteración, siempre y cuando se cumplan algunas condiciones, dadas por los parámetros establecido, los cuales se presentan a continuación:

- ***filepath***, en este parámetro se especifica la ruta del directorio local donde se desea que se guarde el modelo. Este archivo tiene como extensión . h5 y en una sección posterior, severa como cargar y utilizar ese modelo mediante Keras, a partir del archivo guardado.
- ***monitor***, similar a la función de detención temprana, es la variable que se evaluará o monitoreará, para determinar si se guarda o no el modelo de dicha iteración. En este caso, se define la misma variable que en la función anterior, es decir, el valor de función de pérdida en etapa de validación.
- ***save_best_only***, al "setear" este parámetro en True, el último mejor modelo de acuerdo a la cantidad monitoreada no se sobre escribirá.
- ***verbose***, al ser definido como 1, en el proceso de entrenamiento se imprimirá en pantalla cada vez que se sobrescriba el archivo que guarda el modelo.
- ***mode***, de igual forma que en la función anterior, se define como 'min', dado que la situación ideal se presenta cuando la función de pérdida empieza a disminuir y toma valores relativamente cercanos a 0.

Finalmente, se declara la variable callbacks, la cual almacena una lista con las variables referentes a la función de detención temprana y de puntos de control del modelo.

Líneas 58-59:

```
step_size_train=train_generator.n/train_generator.batch_size
step_size_validation=validation_generator.n/validation_generator.bat
      ch_size
```

En estas líneas se calcula el número de muestras por lote que se utilizaran por iteración. En este caso se realiza mediante los generadores de entrenamiento y validación, donde al invocar el atributo. n, se obtiene la cantidad de imágenes cargadas y se divide entre el tamaño del lote, mediante el atributo .batch_size, propio de los generadores.

Línea 60:

```
history=model.fit_generator(generator=train_generator,steps_per_epoc
      h= step_size_train,validation_data = validation_generator,
      validation_steps= step_size_validation,epochs=32,callbacks=
      callbacks)
```

Finalmente, se ha llegado a la línea de código que acciona el entrenamiento del modelo. Esta tarea se realiza mediante la función fit_generator(), a la cual se le pasan los siguientes parámetros:

- ***generator***, es el conjunto de datos de entrenamiento que se cargó mediante el generador de imágenes de Keras, en este caso se utiliza la variable ***train_generator***, la cual almacena el objeto secuencial que contiene las imágenes y el arreglo de etiquetado de las imágenes.
- ***steps_per_epoc***, este parámetro debe ser un entero e indica la cantidad de muestras por lote que se utilizaran por iteración.
  Es allí, donde se hace uso de la variable ***step_size_train***, definida anteriormente.
- ***validation_data***, hace referencia al conjunto de imágenes que se

utilizará para validar el modelo iteración tras iteración. Se utiliza el generador de imágenes de validación que se establecido en pasos anteriores.

- ***validation_steps***, en relación con el parámetro inmediatamente anterior, establece el número de muestras por lote que se utilizará, al hacer uso de este conjunto de datos de validación. Es aquí donde, se implementa la variable ***step_size_validation***.
- ***epochs***, corresponde al número de iteraciones que se disponen para el entrenamiento del modelo. En este caso, se definen un total de 32 iteraciones. Sin embargo, es poco probable que se complete esa cantidad de iteraciones, debido a la función de detención temprana definida.
- ***callbacks***, relacionado con una lista de funciones que se llaman iterativamente cuando se acaba cada una de las iteraciones, debido a esto se declara con la lista nombrada ***callback***, la cual al macena la función de puntos de control del modelo y la detección temprana del entrenamiento.

Además, es relevante exponer que, el proceso de entrenamiento se está almacenado en una variable denominada como history, la cual se utilizará posteriormente en la graficación de la función de pérdida y de las métricas. Por otra parte, al ejecutar esta línea de código se debe presentar por consola un resultado similar al que se expone a continuación (ver figura 37).

```
Epoch 1/32 49/48 [==============================] - 2s 50ms/step - loss: 1.0380 - acc: 0.4265
- val_loss: 1.2350 - val_acc: 0.3083 Epoch 00001: val_loss improved from inf to 1.23499,
saving model to /ruta/local/model.h5

Epoch 2/32 49/48 [==============================] - 1s 28ms/step - loss: 0.8274 - acc: 0.6231
- val_loss: 0.8794 - val_acc: 0.5833 Epoch 00002: val_loss improved from 1.23499 to 0.87940,
saving model to /ruta/local/model.h5

Epoch 3/32 49/48 [==============================] - 2s 35ms/step - loss: 0.7095 - acc: 0.7002
- val_loss: 0.8365 - val_acc: 0.6444 Epoch 00003: val_loss improved from 0.87940 to 0.83650,
saving model to /ruta/local/model.h5
                                    •    •    •
Epoch 20/32 49/48 [==============================] - 2s 35ms/step - loss: 0.1712 - acc: 0.9305
- val_loss: 0.3139 - val_acc: 0.9056 Epoch 00020: val_loss improved from 0.37486 to 0.31392,
saving model to /ruta/local/model.h5

Epoch 21/32 49/48 [==============================] - 2s 35ms/step - loss: 0.1264 - acc: 0.9515
- val_loss: 0.4031 - val_acc: 0.8417 Epoch 00021: val_loss did not improve from 0.31392

Epoch 22/32 49/48 [==============================] - 2s 35ms/step - loss: 0.1320 - acc: 0.9459
- val_loss: 0.3258 - val_acc: 0.8889 Epoch 00022: val_loss did not improve from 0.31392

Epoch 23/32 49/48 [==============================] - 2s 36ms/step - loss: 0.2885 - acc: 0.8923
- val_loss: 0.3339 - val_acc: 0.9093 Epoch 00023: val_loss did not improve from 0.31392

Epoch 24/32 49/48 [==============================] - 2s 36ms/step - loss: 0.1331 - acc: 0.9477
- val_loss: 0.3512 - val_acc: 0.8694 Epoch 00024: val_loss did not improve from 0.31392

Epoch 25/32 49/48 [==============================] - 2s 35ms/step - loss: 0.1202 - acc: 0.9528
- val_loss: 0.3973 - val_acc: 0.8472

Epoch 00025: val_loss did not improve from 0.31392
Epoch 00025: early stopping
```

Figura 37 Iteraciones de entrenamiento.
Fuente: Autor.

De acuerdo con la figura 37, se pueden observar los siguientes aspectos:

- Iteración tras iteración se hace llamada de las funciones de callback, por ejemplo, cuando se enuncia ***saving model to /ruta/local/model.h5*** se hace llamado a la función de puntos de control del model, solo cuando disminuye el ***val_loss***, el cual hace referencia al valor de función de pérdida con el conjunto de validación.
- La función de detención temprana, se ve en acción cuando en ***consola se muestra val_loss improved from y_value to x_value***, donde x_value es menor que y_value. Además, se ve el uso del parámetro ***patience***, dado que, en la iteración número 20 fue la última vez que este valor disminuyó, y durante las siguite cinco iteraciones no se presentó mejora, por lo tanto, en la iteración número 25, se detiene el entrenamiento y el último modelo guardado fue el de la iteración 20.
- Finalmente, se puede observar que, el modelo con el conjunto

de entrenamiento y validación, presenta una función de pérdida relativamente cercana a 0 y una probabilidad de precisión por encima del 0.9.

Líneas 61-67:

```
61. plt.plot(history.history['acc'])
62. plt.plot(history.history['val_acc'])
63. plt.title('model accuracy')
64. plt.ylabel('accuracy')
65. plt.xlabel('epoch')
66. plt.legend(['train', 'test'], loc='upper left')
67. plt.show()
```

Estas líneas de código tienen como objetivo graficar el registro de la métrica accuracy para el dataset de entrenamiento y validación, iteración a iteración. Tenga en cuenta que si usted está utilizando una versión de Tensorflow inferior a 2.0, debe cambiar la claves del diccionario history, por 'acc' y 'val_acc' respectivamente . La figura 38, presenta el resultado de correr estas líneas de código.

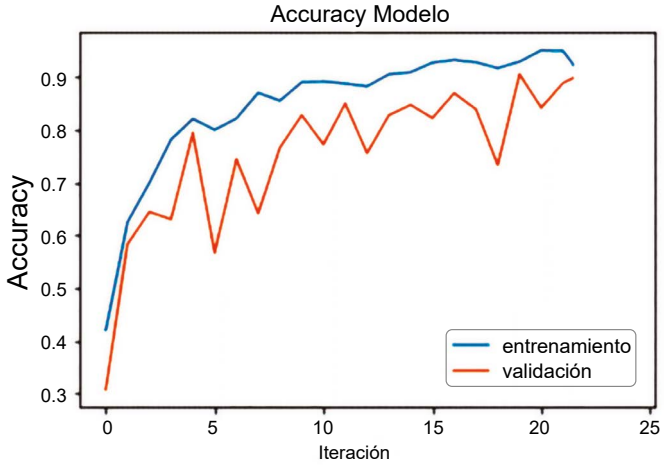


Figura 38 Precisión global del modelo en fase de entrenamiento.
Fuente: Autor.

Líneas 68-74:

```
68. plt.plot(history.history['loss'])
69. plt.plot(history.history['val_loss'])
70. plt.title('model loss')
71. plt.ylabel('loss')
72. plt.xlabel('epoch')
73. plt.legend(['train', 'test'], loc='upper left')
74. plt.show()
```

Mediante estas líneas, se puede graficar el registro de la función de pérdida para el dataset de entrenamiento y validación, iteración a iteración. La figura 39, presenta el resultado de ejecutar estas líneas de código.

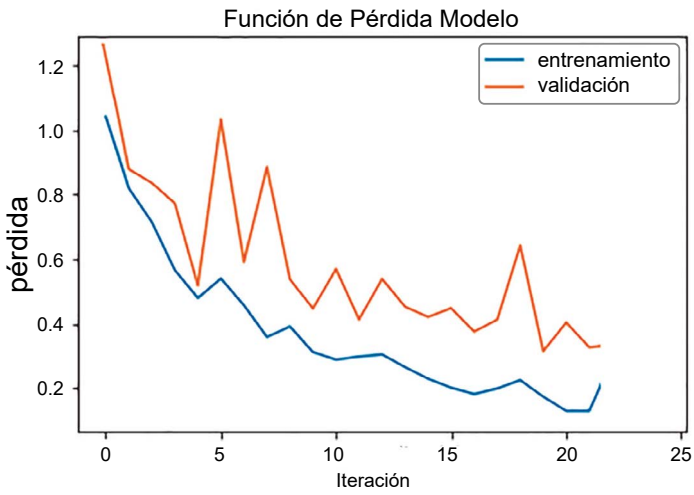


Figura 39 Función de pérdida del modelo en fase de entrenamiento.
Fuente: Autor.

Justo después de obtener un modelo de CNN ya entrenado, es conveniente realizar la evaluación del mismo. Es por esto que, en el siguiente cápitulo se abordará dicho aspecto.

Capítulo 5

Evaluación del Modelo

Una de las tareas más importantes dentro del aprendizaje profundo es poder determinar si el modelo entrenado realiza un buen trabajo en su etapa de clasificación o predicción de clases. Durante esta etapa, se debe establecer un protocolo de evaluación, en el cual se debe tener definido el conjunto de datos de prueba y cuáles van a ser las métricas que se tomarán como punto de referencia para determinar el nivel de rendimiento del modelo de la red neuronal por convolución. En este capítulo, las métricas que se trabajarán son: Función Evalute, Curvas de ROC, Accuracy, Precision Score, Recall Score, Coeficiente de Kappa y la Matriz de confusión. Lo anterior, mediante el conjunto de prueba, compuesto por un total de 300 imágenes, 100 por cada clase.

5.1 Carga del modelo

En el campo del aprendizaje de máquina más específicamente en el aprendizaje profundo, donde el entrenamiento de los modelos de redes neuronales por convolución presenta un alto costo computacional, es necesario poder cargar un modelo que haya sido entrenado previamente. De esta manera, el modelo de la CNN puede ser utilizo en distintos entornos con el mismo propósito, sin la necesidad de entrenarlo de nuevo.

5.1.1 Procedimiento para cargar el modelo

La siguiente línea de código fuente, se utiliza para cargar un modelo de una red neuronal por convolución que haya sido previamente guardado.

```
75. model_loaded = load_model("ruta/local/model.h5")
```

En esta línea se puede observar que, Keras cuenta con una función propia para realizar la carga de un modelo previamente guardado. Esta es la función load_model(), la cual recibe por parámetro, la dirección donde está alojado el modelo y el nombre del archivo, por supuesto, como se había mencionado anteriormente este archivo debe tener una extensión. h5.

En este paso, se carga el modelo entrenado anteriormente, con el propósito de realizar la evaluación de distintas métricas utilizando este modelo. De acuerdo a esto, el modelo cargado se almacena en la variable model_loaded.

5.2 Función Evaluate

Como se presenció durante la fase de entrenamiento del modelo, dos valores son calculados iteración a iteración. Estos valores son: la función de pérdida y el nivel de precisión (accuracy). De acuerdo a esto, la librería Keras, brinda las condiciones necesarias para permitir el cálculo de estos dos valores mediante el conjunto de prueba. Lo anterior, mediante la función evaluate.

5.2.1 Procedimiento para calcular la función evaluate

A continuación, se expone las líneas del código fuente, utilizadas para llevar a cabo el cálculo de la función evaluate.

```
76. step_size_test=test_generator.n/test_generator.batch_size
77. result_evaluate =  model_loaded.evaluate_generator(test_
      generator,step_size_test,verbose=1)
```

Línea 76:

```
step_size_test=test_generator.n/test_generator.batch_size
```

En esta línea se calcula el número de muestras por lote que se utilizaran por iteración. En este caso se realiza mediante el generador de prueba, donde se al invocar el atributo .n, se obtiene la cantidad de imágenes cargadas y se divide entre el tamaño del lote, mediante el atributo .batch_size, propio de los generadores.

Línea 77:

```
result_evaluate =  model_loaded.evaluate_generator(generator=test_
      generator, steps=step_size_test,verbose=1)
```

Finalmente, digitando esta línea se calcula la función de pérdida y la precisión con el conjunto de datos de prueba. Lo anterior, mediante la función evaluate_generator(), la cual recibe lo siguiente parámetros:

- ***generator***, es el conjunto de datos de prueba que se cargó mediante el generador de imágenes de Keras, en este caso se utiliza la variable ***test_generator***, la cual almacena el objeto secuencial que contiene las imágenes y el arreglo de etiquetado de cada imagen.
- ***steps***, es el total de muestras por lote que se estipulan para generar la evaluación del conjunto de datos de prueba.
  En este punto, se utiliza la variable ***step_size_test***, calculada anteriormente.
- ***verbose***, se utiliza tan solo para definir si el proceso y resultado de esta función se observaran en pantalla durante la invocación de la misma. Al definirlo en 1, esto indica que si se presentaran en la consola.

Al ejecutar esta línea se obtiene el siguiente resultado (ver figura 40).

```
300/300 [==============================] - 1s 3ms/step

[0.3224358580739467, 0.8833333333333333]
```

Figura 40 Función Evaluate.
Fuente: Autor.

En la primera línea la figura 40, se presenta el detalle del proceso de la función evaluate_generator(), procesando un total de 300 imágenes en 1, 3 segundos. Después, la segunda línea es el resultado de aplicar esta función. La primera posición de esta lista corresponde al valor de la función de pérdida y la segunda posición al valor de precisión obtenido.

5.3 Curvas de ROC

Una curva ROC (curva de característica operativa del recepto) es una representación gráfica que muestra el rendimiento de un modelo de clasificación en todos los umbrales de clasificación (Zhou, Hall, & Shapiro, 1997), (Zou KH, O'Malley AJ, Mauri L., 2007). Esta curva representa dos parámetros, de un lado la tasa de verdaderos positivos (TPR), y del otro lado la tasa de falsos positivos (FPR).

El punto de partida para el análisis de la curva ROC, es la tabla de contingencia (para validación de imágenes), para cada punto de corte. Esta tabla de contingencia puede entenderse como la matriz de confusión (tabla 1) en la clasificación de imágenes.

Tabla 1 Contingencia o matriz de confusión.
Fuente: Autor.

Matriz de Confusión		Predicho /Modelo	
		Negativo	Positivo
Real	Negativo	VN verdadero negativo	FP Falso Positivo
	Positivo	FN Falso Negativo	VP verdadero positivo

La tasa de verdaderos positivos (TPR) es sinónimo de exhaustividad y, por lo tanto, se define en la ecuación 5:

$$TPR = \frac{VP}{VP + FN}$$

Ecuación 5 Tasa de verdaderos positivos.

Siendo VP los verdaderos positivos, y FN los falsos negativos. Por su parte, la tasa de falsos positivos (FPR) es sinónimo de especificidad, y se define en la ecuación 6:

$$FPR = \frac{FP}{FP+VN}$$

Ecuación 6 Tasa de falsos positivos.

Siendo FP los falsos positivos, y VN los verdaderos negativos.

Tiendo en cuenta que una curva ROC representa la TPR frente a la FPR en diferentes umbrales de clasificación, reducir el umbral de clasificación clasifica más elementos como positivos, por lo que aumentarán tanto los falsos positivos como los verdaderos positivos. La figura 41 muestra una curva ROC típica.

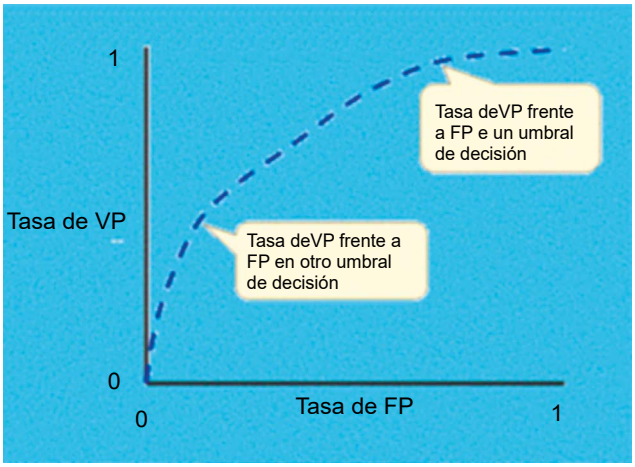


Figura 41 Tasa de VP frente a FP en diferentes umbrales de clasificación. Fuente: Ajustado de (Benavides, 2017).

Para calcular los puntos en una curva ROC, se puede evaluar un modelo de regresión logística muchas veces con diferentes umbrales de clasificación, pero esto es ineficiente. Por lo que se utiliza el algoritmo AUC (área bajo la curva ROC), el cual es eficiente basado en ordenamiento de la información (ver figura 42). El AUC mide toda el área bidimensional por debajo de la curva ROC completa proporciona una medición agregada del rendimiento en todos los umbrales de clasificación posibles de (0,0) a (1,1).

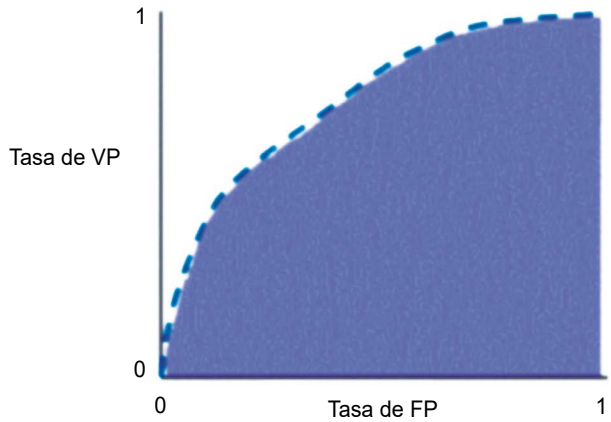


Figura 42 AUC (área bajo la curva ROC).
Fuente: Ajustado de (Benavides, 2017).

El AUC es la probabilidad de que el modelo clasifique un ejemplo positivo aleatorio más alto que un ejemplo negativo aleatorio, oscila en valor del 0 al 1. Un modelo cuyas predicciones son un 100% incorrectas tiene un AUC de 0.0; otro cuyas predicciones son un 100% correctas tiene un AUC de 1.0.En el caso de un AUC de 0,5 es una prueba sin capacidad discriminatoria diagnóstica, de (0.5,0.6) es un test malo, de (0.6,0.75) es un test regular, de (0.75,0.9) es un test bueno, de (0.9,0.97) es un test muy bueno, y de (0.97,1) es un test excelente.

El AUC es conveniente por las dos razones Swets y Picket (1982): la primera es porque es invariable con respecto a la escala ya que mide qué tan bien se clasifican las predicciones, en lugar de sus valores absolutos. La segunda razón es porque el AUC es invariable con respecto al umbral de clasificación, es decir que mide la calidad de las predicciones del modelo, sin tener en cuenta qué umbral de clasificación se elige.

Sin embargo, estas dos razones tienen algunas advertencias, que pueden limitar la utilidad del AUC en determinados casos. Primero la invariabilidad de escala no siempre es conveniente, que el AUC no muestra los resultados de probabilidad bien calibrados. Segundo, la invariabilidad del umbral de clasificación no siempre es conveniente, ya que en los casos en que hay amplias discrepancias en las consecuencias de los falsos negativos

frente a los falsos positivos, es posible que sea fundamental minimizar un tipo de error de clasificación. El AUC no es una métrica útil para este tipo de optimización.

5.3.1 Procedimiento para calcular las curvas de ROC

A continuación, se presenta el código utilizado para el cálculo de las curvas de ROC Macro-promedio y Micro-promedio.

```
78.  y_pred_prob =  model_loaded.predict_generator(generator=test_
      generator, steps= step_size_test)
79.  y_pred_classes = np.argmax(array=y_pred_prob, axis=1)
80.  test_labels_one_hot = to_categorical(test_generator.classes)
81.  fpr = dict()
82.  tpr = dict()
83.  roc_auc = dict()
84.  for i in range(num_classes):
85.      fpr[i], tpr[i], _ = roc_curve(test_labels_one_hot[:, i],
           y_pred_ prob[:, i])
86.      roc_auc[i] = auc(fpr[i], tpr[i])
87. fpr["micro"], tpr["micro"], _=roc_curve(test_labels_one_hot.
      Ravel(), y_pred_prob.ravel())
88. roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])
89. all_fpr = np.unique(np.concatenate([fpr[i] for i in range
      (num_classes)]))
90. mean_tpr = np.zeros_like(all_fpr)
91. for i in range(num_classes):
92.     mean_tpr += interp(all_fpr, fpr[i], tpr[i])
93. mean_tpr /= num_classes
94. fpr["macro"] = all_fpr
95. tpr["macro"] = mean_tpr
96. roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])
```

Líneas 78-80:

```
y_pred_prob =  model_loaded.predict_generator(generator=test_
      generator, steps= step_size_test)
y_pred_classes = np.argmax(array=y_pred_prob, axis=1)
test_labels_categorical= to_categorical(test_generator.classes)
```

Estas líneas de código tienen como finalidad obtener las predicciones realizadas por el modelo a partir del conjunto de datos de prueba. Es importante resaltar que, cada una de las variables declaradas en estas líneas serán utilizadas no solo para calcular las curvas de ROC, si no cada una de las métricas contempladas.

Como primera parte, mediante la función predict_generator(), se generan las predicciones a partir de un conjunto de entradas específicamente un generador de datos. Es decir, por cada imagen evaluada se genera un arreglo con un número de posiciones iguales al número de clases. En dicho arreglo, se consolida la probabilidad que tiene esa imagen de ser cada una de las clases. Donde la posición donde se acumule la mayor probabilidad está directamente relacionada con la clase en la cual el modelo clasificó esa imagen. Los parámetros utilizados para esta función son los siguiente:

- ***generator***, es el conjunto de datos de prueba que se cargó previamente mediante el generador de imágenes de Keras, en este caso se utiliza la variable ***test_generator***, la cual almacena el objeto secuencial que contiene las imágenes y el arreglo de etiquetado de cada imagen.

- ***steps***, es el total de muestras por lote que se estipulan para generar las predicciones del conjunto de datos de prueba. En este punto, se utiliza la variable ***step_size_test***, calculada anteriormente.

Una vez se han calculado las probabilidades de predicción, se procede a obtener puntualmente la clase predicha por el modelo de la red neuronal por convolución. Lo anterior, mediante la función argmax(), propia de

la librería Numpy. Esta función, retorna los índices del elemento máximo de un arreglo sobre un eje particular. En este caso, devuelve la posición del arreglo de predicción de cada imagen, donde se encuentran la mayor probabilidad. Para implementar dicha función, se utilizaron los siguientes parámetros:

- ***array***, en este parámetro se debe especificar el arreglo al cual se le va a aplicar la función. En este caso se iguala a la variable y_pred_prob, que tiene el arreglo de probabilidades generado previamente.

- ***axis***, estipula sobre cual eje se va a obtener el índice donde se encuentra el máximo valor. Este parámetro se define en 1, para aplicar la función de manera horizontal.

Por último, haciendo uso de la función to_categorical(), propia de la librería Keras, se convierten a un formato categórico las etiquetas de clase, guardadas en el generador de imágenes de prueba test_generator.

Líneas 81-83:

```
fpr = dict()
tpr = dict()
roc_auc = dict()
```

En estas líneas, se inicializan diccionarios vacíos para almacenar la tasa de falsos positivos, tasa de verdaderos positivos y el área bajo la curva de ROC. Las variables destinadas para esto fueron fpr, tpr, roc_auc, respectivamente.

Líneas 84-86:

```
for i in range(num_classes):
    fpr[i], tpr[i], _ = roc_curve(y_true=test_labels_categorical[:,
        i], y_score=y_pred_prob[:, i])
    roc_auc[i] = auc(x=fpr[i], y=tpr[i])
```

Este ciclo se usa para iterar en un rango definido por el número clases, en este caso 3. Dentro de este ciclo, se computa la curva de roc y el área bajo cada una de las curvas para cada una de las clases. La función roc_curve() realiza el cálculo de la curva de roc y recibe los siguientes parámetros:

- ***y_true***, hace referencia al conjunto de etiquetas binarias {0,1}. Por lo tanto, son las etiquetas categóricas correctas de las imágenes de prueba, por esa razón, se utiliza la variable ***test_labels_categorical***.
- ***y_score***, define los puntajes objetivos, que pueden ser estimaciones de probabilidad de la clase positiva. Debido a esta razón, se hace uso de la variable ***y_pred_prob***, la cual contienen las probabilidades de predicción de cada una de las imágenes.

Adicionalmente, se calcula el área bajo la curva de ROC de cada una de las clases. Lo anterior, mediante la función auc(), recibiendo los siguientes parámetros.

- ***x***, debe ser un arreglo que contenga las coordenadas de la curva de ROC en el eje X. Por lo tanto, se debe utilizar la variable ***fpr***, la cual contiene la tasa de falsos positivos obtenidos en el cómputo de la curva de ROC.
- ***y***, se refiere al conjunto de coordenadas de la curva de ROC en el eje Y. En este parámetro, se utiliza la variable ***tpr***. Esta variable almacena la tasa de verdaderos positivos obtenidos en el cómputo de la curva de ROC.

Línea 87:

```
fpr["micro"], tpr["micro"], _=roc_curve(y_true=test_labels_one_hot.
      ravel(), y_score=y_pred_prob.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])
```

En estas líneas se crea una nueva clave para los diccionarios de datos creados anteriormente denominada 'micro'. Lo que se pretende con esto, es almacenar los valores asociados a la micro-promedio curva de ROC y

posteriormente, calcular el área bajo de esta. De igual manera que en las líneas 84-86, se utilizan las funciones roc_curve() y auc(). Sin embargo, se presenta una variación, en los argumentos de la función roc_curve(), dado que se utilizan las mismas variables, pero, se les aplica la función propia de la librería Numpy ravel(), la cual convierte un arreglo n-dimensional a un arreglo unidimensional. Lo anterior, con el objetivo de tomar los datos asociados a las etiquetas correctas y de predicción, para procesarlas como un gran conjunto de valores.

Líneas 89-90:

```
all_fpr = np.unique(np.concatenate([fpr[i] for i in range(num_
      classes)]))
mean_tpr = np.zeros_like(all_fpr)
```

Con estas líneas de código se da inicio al proceso de cálculo de la macro-promedio curva de ROC. Como primera parte, mediante la función concatenate() de Numpy, se concatenan los valores de tasa de falsos positivos de cada clase, previamente almacenados en la variable fpr. Posteriormente, se obtienen los valores únicos del arreglo resultante del proceso de concatenación, lo anterior, mediante la función unique(), también de Numpy. Esto se consolida en la variable all_fpr. Adicionalmente, se crea un arreglo lleno de ceros nombrado como mean_tpr, de las mismas dimensiones que all_fpr, para esto se hace uso de la función zeros_like(), pasando por parámetro la variable all_fpr.

Líneas 91-93:

```
for i in range(num_classes):
    mean_tpr += interp(x=all_fpr, xp=fpr[i], fp=tpr[i])
mean_tpr /= num_classes
```

En este ciclo se llena el arreglo mean_tpr, mediante una suma acumulativa. Los valores que se suman acumulativamente, son el resultado de interpolar toda la tasa de falsos positivos, la tasa de falsos positivos por clase y la tasa de verdaderos positivos por clase. Este proceso se realiza

mediante la función interp() propia de la librería Scipy. Dicha función recibe los siguientes parámetros:

- ***x***, son las coordenadas X consolidadas en un arreglo, en las que evaluar los valores interpolados. Para este caso, se debe utilizar la variable ***all_fpr***.
- ***xp***, corresponde a la primera secuencia de números flotantes o coordenadas del eje X, las cuales serán elementos fundamentales en el proceso de interpolación. Como esta actividad se realiza en un ciclo con iteraciones igual al número de clases, se utiliza la variable ***fpr***, siendo indexada por la iteración actual.
- ***fp***, hacer referencia a la segunda secuencia de número flotantes o coordenadas en el eje Y, utilizadas en la tarea de interpolación. Como esta actividad se realiza en un ciclo con iteraciones igual al número de clases, se utiliza la variable ***tpr***, siendo indexada por la iteración actual.

Finalmente, se calcula el promedio de la tasa de verdaderos positivos, dividendo cada uno de los valores alojados en el arreglo mean_tpr sobre el número de clases.

Líneas 94-96:

```
fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])
```

Por último, se crean las claves "macro" en los diccionarios fpr y tpr, almacenando las variables all_fpr y mean_tpr respectivamente. Una vez se ha realizado esta tarea, se computa el área bajo la curva de la macro-promedio de la curva de ROC mediante la función auc() y se guarda su resultado en la clave "macro" del diccionario roc_auc.

5.3.2 Procedimiento para graficar las curvas de ROC

Las siguientes líneas de código fuente, tienen como objetivo realizar una representación gráfica de los valores obtenidos en el procedimiento llevado a cabo para calcular las curvas de ROC.

```
97.   plt.figure(1)
98.   plt.plot((fpr["micro"]), (tpr["micro"]),label='micro-average
      ROC curve (area = {0:0.2f})'''.format(roc_auc["micro"]),color=
      'deeppink', linestyle=':', linewidth=4)
99.   plt.plot(fpr["macro"], tpr["macro"],label='macro-average ROC
      curve (area = {0:0.2f})'''.format(roc_auc["macro"]),color=
      'navy', linestyle=':', linewidth=4)
100.  colors = cycle(['aqua', 'darkorange', 'cornflowerblue'])
101.  for i, color in zip(range(num_classes), colors):
102.      plt.plot(fpr[i], tpr[i], color=color, lw=2,label='ROC
          curve   of class {0} (area = {1:0.2f})'''.format(i,
          roc_auc[i]))
103.  plt.plot([0, 1], [0, 1], 'k—', lw=2)
104.  plt.xlim([0.0, 1.0])
105.  plt.ylim([0.0, 1.05])
106.  plt.xlabel('False Positive Rate')
107.  plt.ylabel('True Positive Rate')
108.  plt.title('Curve ROC General DataSetOriginal')
109.  plt.legend(loc="lower right")
110.  plt.show()
```

Al ejecutar estas líneas, se puede obtener un resultado similar al de la figura 43.

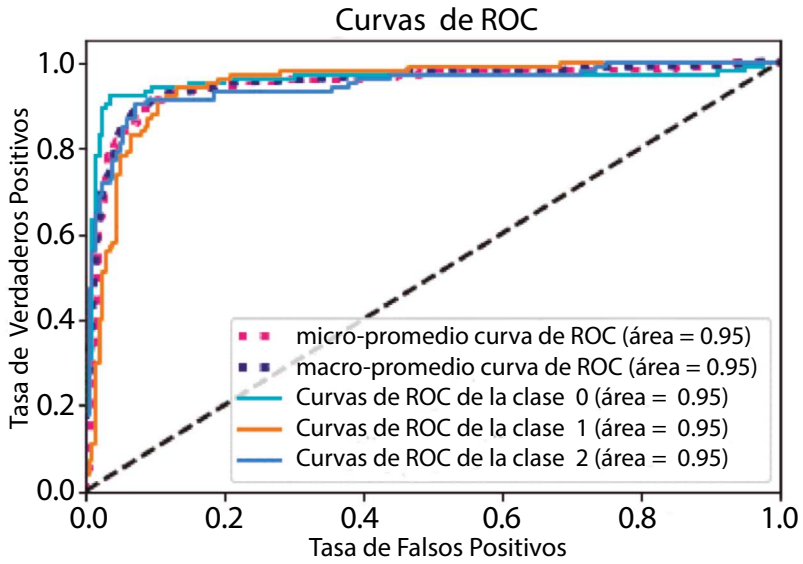


Figura 43 Curvas de ROC del modelo entrenado.
Fuente: Autor.

La figura 43, presenta las Curvas de ROC macro-promedio, micro-promedio con su respectivo valor de área bajo la curva, que en este caso es 0.95 para cada una. Esto indica que presenta en promedio un 0.95 de probabilidad de acierto en su fase de clasificación. Adicionalmente, se presentas las curvas de ROC para cada una de las clases, donde la clase 0 hace referencia a Avión, la clase 1 a Barco y la clase 2 a Estadio.

Por otro lado, Lin, Alvarez y Ruiz (2002) definen las siguientes relaciones: accuracy, precision, y recall.

5.4 Accuracy Score

Se calcula dividiendo el número total de píxeles correctamente clasificados por el número total de píxeles de referencia y expresándolo como porcentaje (ecuación 7).

$$Exactitud\ glabal = \frac{TPC}{TPR}$$

Ecuación 7 Exactitud global.

Donde, TPC es el total de píxeles correctamente clasificados y TPR: corresponde al total de píxeles de referencia.

5.4.1 Procedimiento para calcular el valor de accuracy

Las siguientes líneas de código muestran la forma de calcular la métrica denominada Accuracy.

```
111. accuracy=accuracy_score(y_true=test_generator.classes,y_pred=
      y_pred_classes)
112. print('Accuracy: %f' % accuracy)
```

En estas líneas, se acude a la función accuracy_score(), propia de la librería sckit-learn. Esta función se utiliza para calcular el valor de accuracy y recibe los siguientes parámetros.

- ***y_true***, este parámetro indica las etiquetas correctas del conjunto de imágenes de prueba. Es decir, lo que debería obtener el modelo en la clasificación de cada una de las imágenes. En este caso, se utiliza el atributo .clasess, del generator de imágenes de prueba. Este atributo contiene las etiquetas en su respectivo orden de las imágenes de test cargadas previamente.
- ***y_pred***, hace referencia al conjunto de predicciones realizadas por el modelo, haciendo uso de las imágenes de prueba. Para esta situación, se utiliza la variable y_pred_classes, la cual como se ha mencionado anteriormente contiene las clases predichas por el modelo de la red neuronal por convolución.

Por último, se manda a imprimir el resultado por consola, así como lo muestra la figura 44.

```
Accuracy: 0.790000
```

Figura 44 Métrica de Accuracy (Precisión global).
Fuente: Autor.

5.5 Precision Score

La precisión intenta responder la pregunta ¿Qué proporción de identificaciones positivas fue correcta?, se define en la ecuación 8. Un modelo que no produce falsos positivos tiene una precisión de 1.0.

$$Precisión = \frac{VP}{VP + FP}$$

Ecuación 8 Precisión.

Donde VP son los verdaderos positivos y FP son los falsos positivos.

De manera similar, la exhaustividad intenta responder la pregunta ¿Qué proporción de positivos reales se identificó correctamente?, y se define por la ecuación 9:

$$Exhaustividad = \frac{VP}{VP + FN}$$

Ecuación 9 Exhaustividad

Donde VP son los verdaderos positivos, FN son los falsos negativos.

5.5.1 Procedimiento para calcular el Precision Score

Las siguientes líneas de código muestran la forma de calcular la métrica denominada Precision score.

```
113. precision=precision_score(y_true=test_generator.classes,
      y_pred=y_pred_classes, average='micro')
114. print('Precision:', precision)
```

En estas líneas, se acude a la función precision_score(), propia de la librería sckit-learn. Esta función se utiliza para calcular el valor de precisión y recibe los siguientes parámetros.

- ***y_true***, este parámetro indica las etiquetas correctas del conjunto de imágenes de prueba. Es decir, lo que debería obtener el modelo en la clasificación de cada una de las imágenes.
  En este caso, se utiliza el atributo ***.clasess***, del generator de imágenes de prueba. Este atributo contiene las etiquetas en su respectivo orden de las imágenes de test cargadas previamente.
- ***y_pred***, hace referencia al conjunto de predicciones realizadas por el modelo, haciendo uso de las imágenes de prueba. Para esta situación, se utiliza la variable ***y_pred_classes***, la cual como se ha mencionado anteriormente contiene las clases predichas por el modelo de la red neuronal por convolución.
- ***average***, al ser un problema multiclase, es decir más de una clase, se hace necesario especificar este parámetro. Para le cálculo de esta métrica, se selecciona la opción ***'micro'***, ya que permite realizar un cálculo global contando el total de verdaderos positivos, falsos negativos y falsos positivos.

Por último, se manda a imprimir el resultado por consola,así como lo muestra la figura 45.

```
Precision: 0.790113
```

Figura 45 Métrica Precision Score.
Fuente: Autor.

5.6 Recall Score

El recall score es intuitivamente la capacidad del clasificador para encontrar todas las muestras positivas. El mejor valor es 1 y el peor valor es 0. Se calcula según la ecuación 10.

$$Recall = \frac{TP}{TP + FN}$$

Ecuación 10 Recall

Donde TP es el número de verdaderos positivos y FN el número de falsos negativos.

5.6.1 Procedimiento para calcular el Recall Score

Las siguientes líneas de código muestran la forma de calcular la métrica denominada Recall score.

```
115. recall = recall_score(y_true=test_generator.classes, y_pred=
      y_pred_classes, average = 'micro')
116. print('Recall: %f' % recall)
```

Estas líneas tienen como objetivo, obtener el valor de Recall. Lo anterior, mediante la función recall_score(), propia de la librería Scikit-learn. Esta función recibe los siguientes parámetros.

- ***y_true***, este parámetro indica las etiquetas correctas del conjunto de imágenes de prueba. Es decir, lo que debería obtener el modelo en la clasificación de cada una de las imágenes. En este caso, se utiliza el atributo ***.clasess***, del generator de imágenes de prueba. Este atributo contiene las etiquetas en su respectivo orden de las imágenes de test cargadas previamente.
- ***y_pred***, hace referencia al conjunto de predicciones realizadas por el modelo, haciendo uso de las imágenes de prueba. Para esta situación, se utiliza la variable ***y_pred_classes***, la cual como se ha mencionado anteriormente contiene las clases predichas por el modelo de la red neuronal por convolución.
- ***average***, al ser un problema multiclase, es decir más de una clase, se hace necesario especificar este parámetro. Para le cálculo de esta métrica se selecciona la opción ***'micro'***, ya que permite realizar un cálculo global contando el total de verdaderos positivos, falsos negativos y falsos positivos.

Por último, se manda a imprimir el resultado por consola (ver figura 46).

```
Recall: 0.790000
```

Figura 46 Métrica Recall Score.
Fuente: Autor.

5.7 F1 Score

F1-Score (también llamado Valor-F o Medida-F en español) (Lewis y Gale, 1994) combina las medidas de precisión y exhaustividad para devolver una medida de calidad más general del modelo. Se calcula como la media armónica de las métricas mencionadas (ecuación 11):

$$F1-score = \frac{2}{\frac{1}{precisión} + \frac{1}{exhaustividad}} = \frac{TP}{TP + \frac{FP + FN}{2}}$$

Ecuación 11 F1 - Score.

El valor del F1-Score varía entre 0 (peor valor posible) y 1 (mejor valor posible).

Concejero (2004) concluye que las curvas ROC empíricas son una herramienta potente, y que, con el esquema de contraste de hipótesis estadísticas sin supuestos de partida en cuanto a la forma de las distribuciones, son una metodología a la vez sencilla y potente que ha visto reflejadas estas características en la enorme cantidad de estudios que la utilizan.

5.7.1 Procedimiento para calcular F1 Score

Las siguientes líneas de código muestran la forma de calcular la métrica denominada F1 score.

```
117. f1 = f1_score(y_true=test_generator.classes,y_pred=y_pred_
      classes,average= 'micro')
118. print('F1 score: %f' % f1)
```

En estas líneas, se hace uso de la función f1_score(), propia de la librería Scikit-learn. Esta función se utiliza para calcular el valor de esta métrica y recibe los siguientes parámetros.

- ***y_true***, este parámetro indica las etiquetas correctas del conjunto de imágenes de prueba. Es decir, lo que debería obtener el modelo en

la clasificación de cada una de las imágenes. En este caso, se utiliza el atributo .clasess, del generator de imágenes de prueba. Este atributo contiene las etiquetas en su respectivo orden de las imágenes de test cargadas previamente.

- ***y_pred***, hace referencia al conjunto de predicciones realizadas por el modelo, haciendo uso de las imágenes de prueba. Para esta situación, se utiliza la variable ***y_pred_classes***, la cual como se ha mencionado anteriormente contiene las clases predichas por el modelo de la red neuronal por convolución.
- ***average***, al ser un problema multiclase, es decir más de una clase, se hace necesario especificar este parámetro. Para el cálculo de esta métrica, se selecciona la opción ***'micro'***, ya que permite realizar un cálculo global contando el total de verdaderos positivos, falsos negativos y falsos positivos.

Finalmente, se manda a imprimir el resultado por consola, así como lo muestra la figura 47.

```
F1 score: 0.788318
```

Figura 47 Métrica F1 Score.
Fuente: Autor.

5.8 Coeficiente de Kappa

El índice de Kappa, un instrumento diseñado por Cohen que ajusta el efecto del azar en la proporción de la concordancia observada. La estimación por el índice de Kappa sigue la ecuación:

$$K = \frac{P_o - P_e}{1 - P_e}$$

Ecuación 12 Coeficiente de Kappa.

Donde Po es la proporción de concordancia observada, Pe es la proporción de concordancia esperada por azar y 1 – Pe, representa el acuerdo o concordancia máxima posible no debida al azar. Entonces, el numerador del coeficiente Kappa expresa la proporción del acuerdo observado

menos el esperado, en tanto que el denominador es la diferencia entre un total acuerdo y la proporción esperada por azar (J. Wang, Y. Yang and B. Xia, 2019). En conclusión, el Kappa corrige el acuerdo sólo por azar, en tanto es la proporción del acuerdo observado que excede la proporción por azar. Si este valor es igual a 1, estaríamos frente a una situación en que la concordancia es perfecta (100% de acuerdo o total acuerdo) y, por tanto, la proporción por azar es cero; cuando el valor es 0, hay total desacuerdo y entonces la proporción esperada por azar se hace igual a la proporción observada.

(Landis y Koch, 1977) propusieron una interpretación cualitativa del índice de Kappa utilizada clásicamente en la que la fuerza de concordancia se califica como:

- Pobre o débil para valores menores a 0,40,
- Moderada, para valores de entre 0,41 y 0,60,
- Buena, entre 0,61 y 0,80, y
- Muy buena para valores superiores hasta 1. (Altman, 1991)

5.8.1 Procedimiento para calcular el coeficiente de Kappa

Las siguientes líneas de código muestran la forma de calcular la métrica denominada Coeficiente de Kappa.

```
119. kappa=cohen_kappa_score(y1=test_generator.classes, y2=y_pred
      _classes)
120. print('Cohens kappa: %f' % kappa)
```

Con el propósito de obtener el coeficiente de Kappa, se implementa la función cohen_kappa_score(), la cual hace parte de la librería Scikit-learn. Dicha función utiliza los siguientes parámetros.

- ***y1***, es el conjunto de etiquetas, las cuales van a ser el punto de comparación de otro conjunto de etiquetas. Para este caso, nuestro conjunto de etiquetas referente son las clases del generador de prueba, obteniéndolas a partir del atributo ***. classes*** de la variable ***test_generator***.

- **y2**, hace referencia al segundo conjunto de etiquetas, es decir, el que se quiere comparar frente a los datos de referencia. De acuerdo a lo anterior, este parámetro se iguala al arreglo que contiene las predicciones realizadas por el modelo, por lo tanto, invocamos la variable ***y_pred_classes***.

Al ejecutar estas líneas de código, se debería presentar un resultado similar al presentado en la figura 48.

```
Cohens Kappa: 0.685000
```

Figura 48 Métrica Coeficiente de Kappa.
Fuente: Autor.

5.9 Matriz de Confusión

El contenido de una matriz de confusión es un conjunto de valores que contabilizan el grado de semejanza entre observaciones emparejadas: un conjunto de datos bajo control (CDC) y un conjunto de datos de referencia (CDR), para los que se ha establecido una clasificación. Usualmente el CDR es la verdad terreno, es decir, la realidad, y suele conocerse por medio de un muestreo. La matriz de confusión puede construirse a partir de píxeles, agrupaciones de píxeles o cualquier tipo de objeto geográfico (p.ej.polígonos). Con independencia de su tipología, los elementos del CDC se comparan con sus homólogos en el CDR.

Se trata de una matriz cuadrada de dimensión M´M (filas ´ columnas), donde M denota el número de clases en consideración. Las clases del CDR las denominamos (clases referencia) Γ_R y las clases del CDC las denominamos (clases producto) G_P. Cada uno de los M2 elementos de la matriz los denominamos celdas de la matriz. Las celdas de la diagonal de la matriz de confusión contienen las cantidades correspondientes a los ítems bien clasificados (coincide una G_P con su correspondiente Γ_R). Estas celdas las denominamos C_C (celdas coincidencia). Las celdas de fuera de la diagonal contienen las cantidades correspondientes a las confusiones, los errores debidos a las omisiones y comisiones. Estas celdas las denominamos C_e (celdas error o no coincidencia). Como se puede ver en la tabla 2, sección 5.3.

5.9.1 Procedimiento para calcular la matriz de confusión

A continuación, se presenta el código fuente para computar los valores de la matriz de confusión.

```
121. matrix=confusion_matrix(y_true=test_generator.classes,y_pred=
      y_pred_classes)
122. print(matrix)
```

En estas líneas, se hace uso de la función confusión_matrix(), de la librería Scikit-learn. Esta función se utiliza para calcular el valor de esta métrica y recibe los siguientes parámetros.

- ***y_true***, este parámetro indica las etiquetas correctas del conjunto de imágenes de prueba. Es decir, lo que debería obtener el modelo en la clasificación de cada una de las imágenes. En este caso, se utiliza el atributo .clasess, del generator de imágenes de prueba. Este atributo contiene las etiquetas en su respectivo orden de las imágenes de test cargadas previamente.
- ***y_pred***, hace referencia al conjunto de predicciones realizadas por el modelo, haciendo uso de las imágenes de prueba. Para esta situación, se utiliza la variable ***y_pred_classes***, la cual como se ha mencionado anteriormente contiene las clases predichas por el modelo de la red neuronal por convolución.

Al ejecutar estas líneas de código, se presentaría un resultado por consola como el siguiente (ver figura 49).

```
[[ 87  10   3]
 [  4  81  15]
 [ 16  15  69]]
```

Figura 49 Métrica Matriz de Confusión.
Fuente: Autor.

5.9.2 Procedimiento para graficar la matriz de confusión

Las siguientes líneas de código, tiene como objetivo presentar de forma agradable y gráfica la matriz de confusión.

```
123. cmap = plt.get_cmap('Blues')
124. plt.figure(figsize=(8, 6))
125. plt.imshow(matrix, interpolation='nearest',cmap=cmap)
126. plt.title("Confusion Matrix")
127. plt.colorbar()
128. target_names = ['avión', 'barco', 'estadio']
128. tick_marks = np.arange(len(target_names))
129. plt.xticks(tick_marks, target_names, rotation=45)
130. plt.yticks(tick_marks, target_names)
131. thresh = matrix.max() / 1.5
132. matrix.max() / 2
133. for i, j in itertools.product(range(matrix.shape[0]), range
      (matrix.shape[1])):
134.       plt.text(j, i, "{:,}".format(matrix[i, j]), horizont
            alalignment="center",color="white" if matrix[i, j] >
            thresh else "black")
135. plt.tight_layout()
136. plt.ylabel('Etiqueta Verdadera')
137. plt.xlabel('Etiqueta predicha ')
138. plt.show()
```

Al ejecutar estas líneas, se obtiene un resultado similar al presentado a continuación (ver figura 50).

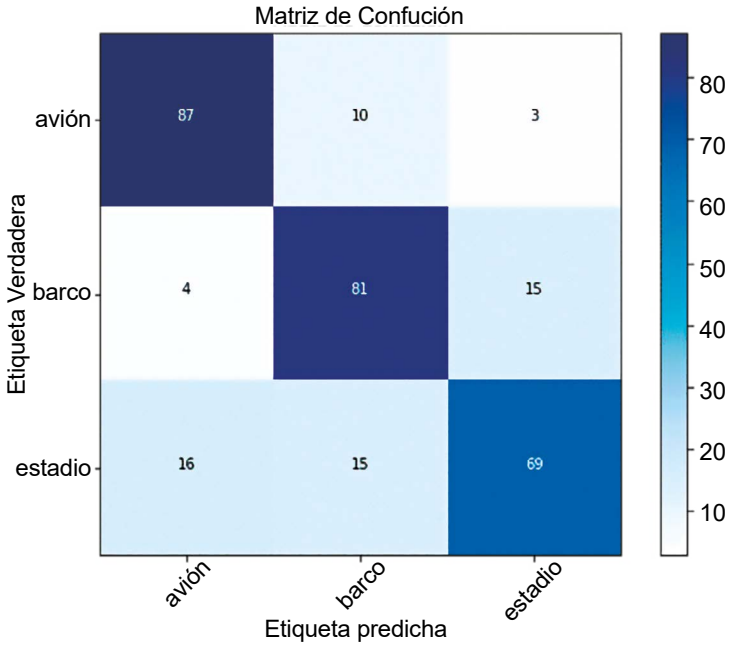


Figura 50 Resultado gráfico Matriz de Confusión.
Fuente: Autor.

Con este capítulo, se finaliza el proceso de codificación dispuesto en este libro. A continuación se presentan lo resultados obtenidos durante el desarrollo de esta investigación y su respectivo análisis.

Capítulo 6

Resultados y Análisis

En esta sección se presentan y analizan los resultados de la evaluación y comparación de los tiempos de entrenamiento del modelo implementado en los capítulos anteriores frente a modelos típicos pre-entrenados como: MobileNet, MobileNetV2, ResNet50 y VGG16. La evaluación se realiza tanto en CPU como en GPU para medir el Speed-up de cada modelo y entre modelos.

Este capítulo tiene la siguiente estructura. Primero, se presentan las características del entorno de prueba donde se llevó a cabo el entrenamiento de los distintos modelos. Luego, se presenta una comparación de tiempos general entre todos los modelos de redes neuronales por convolución. Por último, se exponen resultados específicos por cada uno de los modelos, presentando los tiempos obtenidos iteración a iteración.

6.1 Entorno de prueba

La figura 51, presenta las características más relevantes del entorno donde se realizó el entrenamiento de los modelos. Dentro de estos aspectos, se encuentra el uso de un computador con sistema operativo Win-

dows10. Adicionalmente, este computador cuenta con una tarjeta graficadora GeForce GTX 1070, un procesador Intel(R) Core(TM) I9 de octava generación y 16GB de memoria RAM. A partir de los anterior, se puede argumentar que tanto para el procesamiento en CPU y GPU, se cuentan con excelentes características, cerrando la posibilidad ventajas en algunos de los dos casos.

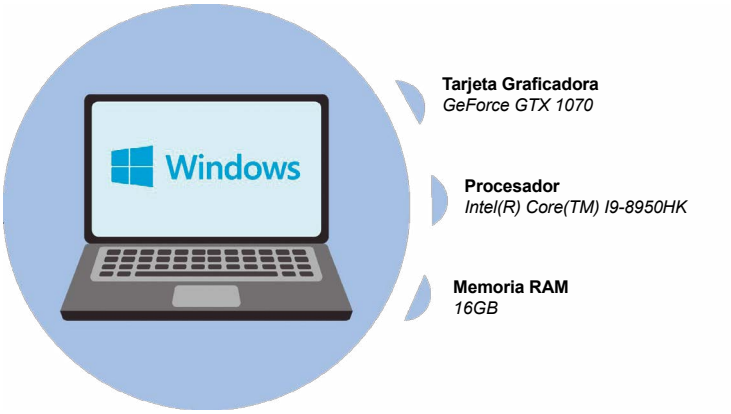


Figura 51 Entorno de prueba.
Fuente: Autor.

6.2 Resultado General

Esta subsección presenta una comparación general cada de uno de los modelos, sin discriminar el número total de iteraciones de entrenamiento, tan solo tomando en cuenta el tiempo total de entrenamiento. La tabla 2, presenta los modelos, su tiempo de ejecución en CPU y en GPU, y la aceleración obtenida al dividir el tiempo de CPU sobre el tiempo de GPU.

Tabla 2 Tiempo de ejecución general.
Fuente: Autor.

Modelo	Tiempo en CPU (s)	Tiempo en GPU (s)	Aceleración
Implementado	308,01	46,16	6,67x
MobileNet	337,82	24,28	13,91x
MobileNetV2	442,8	33,72	13,13x
ResNet50	1122,4	57,13	19,65x
VGG16	1029,4	45,60	22,57x

Adicionalmente, la figura 52, expone gráficamente los datos consolidados en la tabla 2.

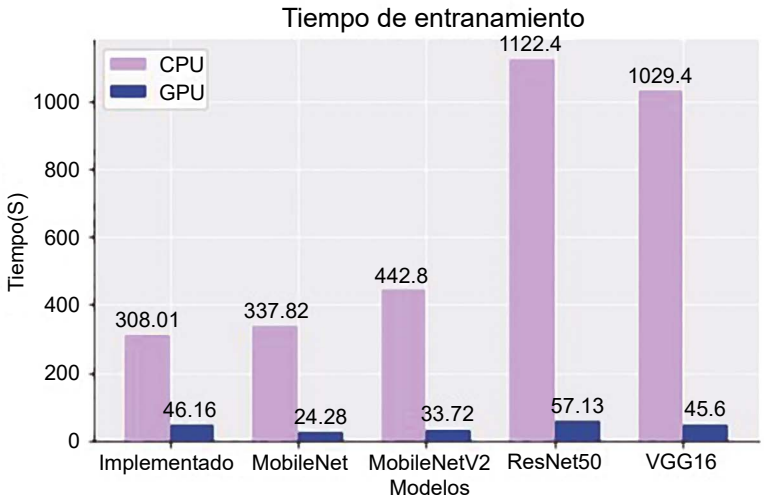


Figura 52 Comparación general de tiempos de entrenamiento de todos los modelos de CNN. Fuente: Autor.

6.3 Resultados específicos

Esta subsección exhibe los tiempos de entrenamiento para cada una de las iteraciones requeridas por cada modelo utilizado. Estos datos, se consolidan en una tabla y posterior se presentan de una manera gráfica.

6.3.1 Modelo Implementado

Para el modelo implementado, como se presentó en el Capítulo 4 apartado 4.3, el entrenamiento del modelo tomó un total de 20 iteraciones. Es por esto, que la tabla 3, presenta el tiempo de ejecución de cada iteración al realizar el entrenamiento en CPU y GPU.

Tabla 3 Tiempo de entrenamiento por iteraciones Modelo Implementado.
Fuente: Autor.

	Dispositivo			Dispositivo	
No. Iteración	**CPU**	**GPU**	**No. Iteración**	**CPU**	**GPU**
1	13.34	2.5	**11**	15.30	2.36
2	15.31	1.28	**12**	16.32	2.35
3	15.31	2.35	**13**	15.30	2.35
4	15.31	2.35	**14**	15.31	2.35
5	15.30	2.35	**15**	15.30	2.37
6	15.31	2.37	**16**	15.31	2.37
7	16.32	2.35	**17**	16.31	2.36
8	16.32	2.35	**18**	15.31	2.35
9	15.31	2.35	**19**	15.31	2.35
10	15.31	2.35	**20**	15.31	2.35
	Tiempo (s)			**Tiempo (s)**	

Además de esto, la figura 53, expone el tiempo de entrenamiento acumulado iteración tras iteración del modelo implementado.

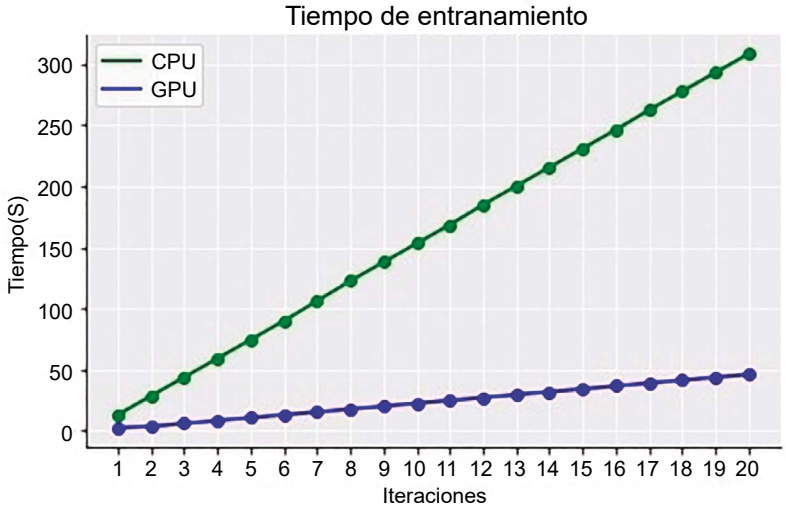


Figura 53 Tiempo de entrenamiento iteración a iteración del modelo Implementado.
Fuente: Autor.

6.3.2 Modelos Pre-entrenados

En esta sección, se presenta la comparación de cuatro modelos de CNN típicos los cuales presentan una arquitectura robusta. Lo anterior, con el propósito de observar al comportamiento del entrenamiento en CPU y CPU. Dado que la estructura de los modelos de redes neuronales convolucionales, es uno de los factores que puede incidir en el tiempo de entrenamiento.

Adicionalmente, para el entrenamiento de estos modelos solo se utilizaron 8 iteraciones, ya que, de acuerdo con (Sarkar, D., Bali, R., & Ghosh, T., 2018) al entrenar estos modelos no se requiere un gran número de iteraciones, convirtiéndose en una ventaja de la transferencia de aprendizaje. A continuación de presentan cada uno de los modelos.

6.3.2.1 MobileNet

La tabla 4, expone el tiempo de entrenamiento utilizado en cada una de las iteraciones para el modelo MobileNet.

Tabla 4 Tiempo de entrenamiento por iteraciones MobileNet.
Fuente: Autor.

	Iteraciones							
Dispositivo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
CPU	44,90	41,85	41,85	41,84	41,85	41,84	41,85	41,84
GPU	7,13	2,45	2,45	2,45	2,45	2,45	2,45	2,45
	Tiempo (s)							

De igual manera, la figura 54, presenta el tiempo de entrenamiento acumulado iteración tras iteración del modelo MobileNet.

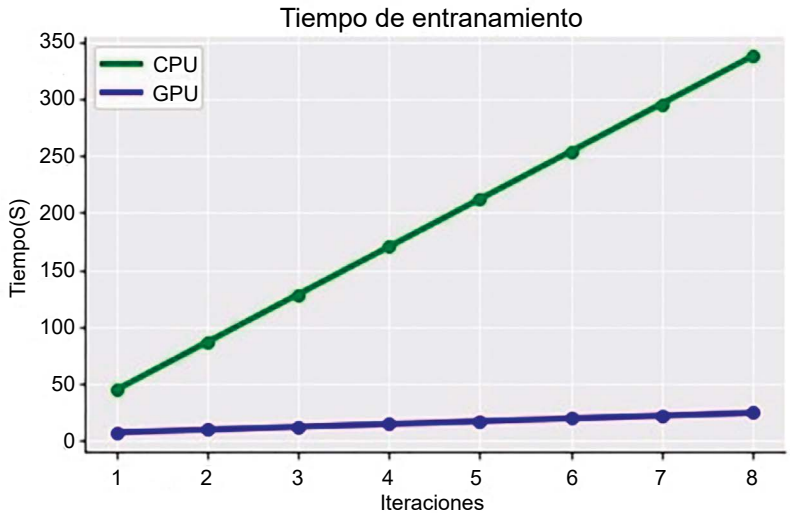


Figura 54 Tiempo de entrenamiento iteración a iteración de MobileNet.
Fuente: Autor.

6.3.2.2 MobileNetV2

La tabla 5, presenta el tiempo de entrenamiento utilizado en cada una de las iteraciones para el modelo MobileNetV2.

Tabla 5 Tiempo de entrenamiento por iteraciones MobileNetV2.
Fuente: Autor.

	Iteraciones							
Dispositivo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
CPU	60,1	56,1	55,1	55,1	55,1	54,1	54,1	53,1
GPU	8,17	3,65	3,65	3,65	3,65	3,65	3,65	3,65
	Tiempo (s)							

Asimismo, la figura 55, expone el tiempo de entrenamiento acumulado iteración tras iteración del modelo MobileNetV2.

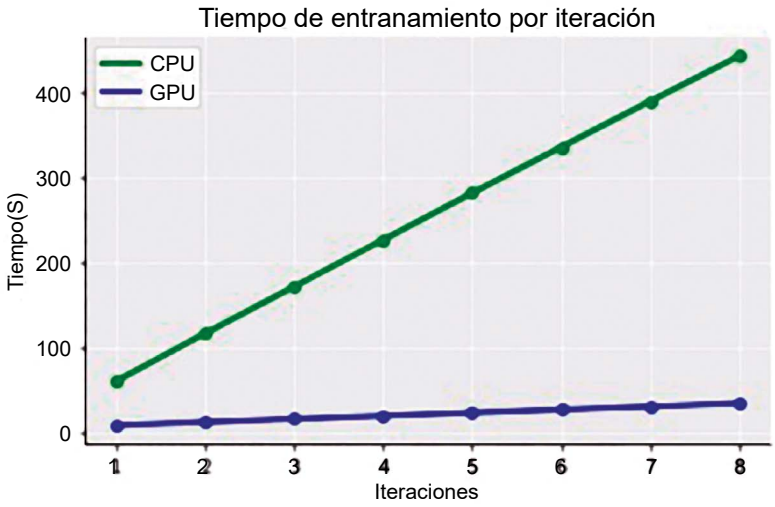


Figura 55 Tiempo de entrenamiento iteración a iteración de MobileNetV2.
Fuente: Autor.

6.3.2.3 ResNet50

La tabla 6, presenta los tiempos de entrenamiento utilizado en cada una de las iteraciones para el modelo MobileNetV2.

Tabla 6 Tiempo de entrenamiento por iteraciones ResNet50.
Fuente: Autor.

	Iteraciones							
Dispositivo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
CPU	143,3	139,3	140,3	140,3	139,3	137,3	140,3	142,3
GPU	14,28	6,12	6,12	6,12	6,12	6,12	6,12	6,12
	Tiempo (s)							

También, la figura 56, exhibe el tiempo de entrenamiento acumulado iteración tras iteración del modelo ResNet50.

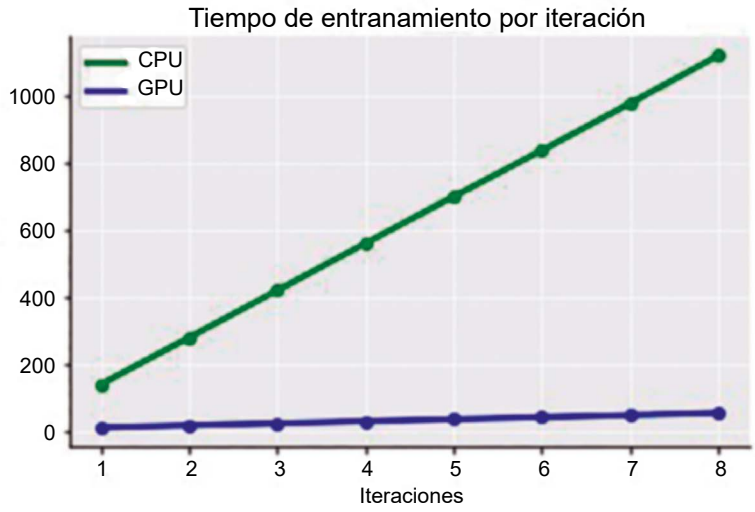


Figura 56 Tiempo de entrenamiento iteración a iteración de ResNet50. Fuente:Autor.

6.3.2.4 VGG16

La tabla 7, expone los tiempos de entrenamiento utilizado en cada una de las iteraciones para el modelo VGG16.

Tabla 7 Tiempo de entrenamiento por iteraciones VGG16. Fuente: Autor.

	Iteraciones							
Dispositivo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
CPU	129,3	123,3	125,3	131,3	128,3	131,3	129,3	131,3
GPU	8,154	5,93	5,94	4,91	5,93	4,91	4,91	4,92
	Tiempo (s)							

Además, la figura 57, presenta el tiempo de entrenamiento acumulado iteración tras iteración del modelo VGG16.

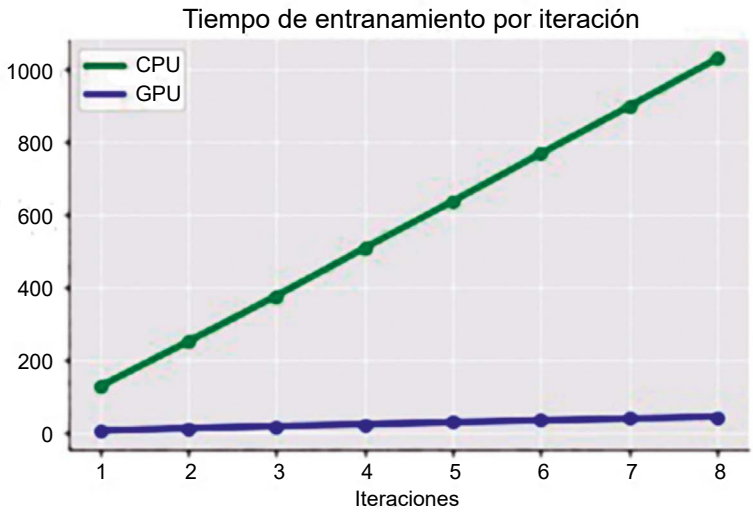


Figura 57 Tiempo de entrenamiento iteración a iteración de VGG16. Fuente: Autor.

6.4 Análisis

Los resultados obtenidos en cuanto al factor de aceleración general (speed-up) para cada uno de los modelos evaluados evidencian que las plataformas many-threads y en este caso específico las GPUs, representan una solución real y eficiente a la intensividad del proceso de entrenamiento de las técnicas de aprendizaje profundo, como lo son las redes neuronales convolucionales. El factor de aceleración osciló entre 6,67x (para el modelo implementado) y 22,57x (para el modelo VGG16) lo que representa que en el peor de los casos el tiempo de entrenamiento se reduce en un 85% y en el mejor de los casos se reduce casi un 96%. Los modelos que presentaron mejor factor de aceleración fueron precisamente aquellos modelos que requieren mayor tiempo de entrenamiento, ResNet50 pasó de emplear 1122,4 segundos sobre CPU a solo tomar 57,13 segundos cuando se realizó sobre GPU y VGG16 pasó de emplear 1029,4 segundos sobre CPU a 45,6 segundos cuando se realizó sobre GPU.

Los resultados de la evaluación discriminados por época evidencian un patrón en el factor de aceleración muy similar en todos los modelos medidos: la primera época de entrenamiento presenta el menor factor de

aceleración, mientras que las siguientes épocas presentan un mejor factor de aceleración que se mantiene relativamente estable época a época. Donde más se evidencia esa diferencia entre la primera época y el resto, es en el modelo MobileNet el cual presenta un factor de aceleración de 6,3x para la primera época de entrenamiento y un valor cercano a 17,08x para el resto de épocas. Los modelos MobilNet en sus dos versiones y el modelo ResNet50 presentan una particularidad, y es que los tiempos de entrenamiento sobre GPU para las épocas diferentes a la primera se mantienen totalmente estables, a pesar de que los tiempos sobre CPU para las mismas épocas varían levemente. En el caso de MobilNet los tiempos de entrenamiento para todas las épocas diferentes a la primera fueron de 2,45 segundos, para la segunda versión de este modelo los tiempos para esas épocas fueron iguales a 3,65 segundos y para el modelo ResNet50 fueron de 6,12 segundos.

Conclusiones

El aprendizaje profundo fundamenta sus buenos niveles de precisión en tareas de predicción o de clasificación en la capacidad de identificación y extracción automática de rasgos mediante un proceso de abstracción jerárquica e iterativa basada en operaciones tales como la convolución y el agrupamiento en las redes neuronales convolucionales; operaciones que aunque su complejidad es media o baja, durante el entrenamiento se convierten en un reto computacional debido a dos factores: su aplicación se basa en un barrido a través de todos los elementos de cada dato de entrada y el tamaño de los conjuntos de datos de entrenamiento normalmente tienen un volumen categorizado como big data debido a que el éxito del proceso de entrenamiento depende en gran medida del tamaño del conjunto de datos. El análisis comparativo de los resultados de la evaluación de los tiempos de entrenamiento tanto del modelo implementado como de los modelos típicos sobre los dos tipos de plataforma, multi-core (CPU) y many-thread (GPU) evidenció que la computación paralela sobre GPU representa una solución eficiente y al reto computacional que implica el entrenamiento de modelos de aprendizaje profundo tales como las redes neuronales convolucionales. Los resultados a niveles de speed-up obtenidos en este libro represetan un marco de referencia de desempeño en cuanto a recursos computacionales muy útil en el proceso de compara-

ción y selección de modelos de redes neuronales convolucionales tanto de forma global como de forma detallada por iteraciones.

Mediante el desarrollo de la solución del caso de estudio afrontado en este libro se pudo evidenciar que el proceso de diseño, implementación, entrenamiento y evaluación de modelos de redes neuronales convolucionales se puede llevar a cabo de forma ágil y fácil utilizando una API de alto nivel como lo es Keras. Esta API ofrece las utilidades necesarias para acondicionar y/o aumentar conjuntos de datos de entrenamiento y de prueba, para construir modelos con las principales funciones y operaciones que involucran las diferentes capas de una red neuronal convolucional, para compilar y entrenar modelos tanto en CPU como en GPU con una configuración muy sencilla, para evaluar y analizar el desempeño de los modelos generados.

Referencias

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016). Tensorflow: A system for large-scale machine learning. In 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16) (pp. 265-283).

Alba, E. (2005). Parallel metaheuristics: a new class of algorithms (Vol. 47). John Wiley & Sons.

Altman DG.(1991) Practical statistics for medical research. New York: Chapman and Hall/CRC. p. 277-300

Amar Shan. (2006). Heterogeneous processing: a strategy for augmenting moore's law. Linux J. 2006, 142, 1-7

ASPRS - American Society of Photogrammetry. (1980). Manual of Photogrammetry, fourth edition, 866 p.

Ariza-López F. J., Rodríguez-Avi, J., Alba-Fernández, V. (2018): "Control estricto de matrices de confusión por medio de distribuciones multinomiales", GeoFocus (Artículos), nº 21, p. 215-226. ISSN: 1578-5157 http://dx.doi.org/10.21138/GF.591

Benavides A. R del V. (2017). Curvas ROC (Receiver-Operating-Characteristic)y sus aplicaciones. Universidad de Sevilla. URI: http://hdl.handle.net/11441/63201.

Burachonok, V. (12 de Enero de 2017). Keras for search ships in satellite Image. Obtenido de https://www.kaggle.com/byrachonok/keras-for-search-ships-in-satellite-image.

Cheng, G., Han, J., & Lu, X. (2017). Remote sensing image scene classification: Benchmark and state of the art. Proceedings of the IEEE, 105(10), 1865-1883.

Chollet, F. (2015). Keras.

Cohen J. (1960). A coefficient of agreement for nominal scales. Educational and psychological measurement. 20: 37-46.

Concejero P, (2004). Comparación de modelos de curvas ROC para la evaluación de procedimientos estadísticos de predicción en investigación de mercados, Tesis doctoral, Universidad Complutense de Madrid.

Cortes, Corinna; Vapnik, Vladimir N. (1995). "Support-vector networks". Machine Learning. 20 (3): 273–297. doi:10.1007/BF00994018.

De Antonio, M., & Marina, L. (2005). Computación paralela y entornos heterogéneos.

Giovanni, G., & Velasquez, V. (2014). Una aproximación conceptual a la implementación de árboles de decisión en la minería de datos espaciales.

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. Computing in science & engineering, 9(3), 90.

Jones, E., Oliphant, T., & Peterson, P. (2001). SciPy: Open source scientific tools for Python.

Kirk, D. B., & Wen-mei, W. H. (2012). Programming massively parallel processors: a hands-on approach. Newnes.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

Landis JR, Koch GG. (1977). The measurement of observer agreement for categorical data. Biometrics Mar;33:159-74.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. nature, 521(7553), 436-444.

Lewis, D., & Gale, W.A. (1994). Training text classifiers by uncertainty sampling. SIGIR 1994.

Lin, W., Alvarez, S. A., Ruiz, C. (2002): Efficient Adaptive-Support Association rule mining for recommender systems. Data Mining and Knowledge Discovery, 6, 83-105.

Mather Paul, Tso Brandt. (2009). Classification Methods for Remotely Sensed Data. 2nd Edition. CRC Press. ISBN 9781420090727.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. Journal of machine learning research, 12(Oct), 2825-2830.

R. González, R. Woods. (1996). Tratamiento Digital de Imágenes. Ed, Eddison-Wesley. EE.UU.

Restrepo Rodríguez, A. O., Casas Mateus, D. E., García, G., Alonso, P., Montenegro Marín, C. E., & González Crespo, R. (2018). Hyperparameter Optimization for Image Recognition over an AR-Sandbox Based on Convolutional Neural Networks Applying a Previous Phase of Segmentation by Color–Space. Symmetry, 10(12), 743.

Sarkar, D., Bali, R., & Ghosh, T. (2018). Hands-On Transfer Learning with Python: Implement advanced deep learning and neural network models using TensorFlow and Keras. Packt Publishing Ltd.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

Swets, J. A. y Pickett, R. M. (1982): Evaluation of Diagnostic Systems. Methods from Signal Detection Theory. Nueva York: Academic Press.

Torres, J. (2018): Deep Learning Introducción práctica con Keras, Barcelona: Serie: Watch This Space. libro (4). ISBN 978-1-983-12981-0.

Van Der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering, 13(2), 22.

Wang, J, Yang, Y. and Xia, B.. (2019). A Simplified Cohen's Kappa for Use in Binary Classification Data Annotation Tasks. in IEEE Access, vol. 7, pp. 164386-164397.

Wu, Jian & Cui, Zhiming & Sheng, Victor & Shi, Yujie & Zhao, Pengpeng. (2014). Mixed Pattern Matching-Based Traffic Abnormal Behavior Recognition. TheScientificWorldJournal. 2014. 834013. 10.1155/2014/834013.

Yang, Y., & Newsam, S. (2010, November). Bag-of-visual-words and spatial extensions for land-use classification. In Proceedings of the 18th SIGSPATIAL international conference on advances in geographic information systems (pp. 270-279). ACM.

Zhou X.H, Hall W.J, Shapiro, D.E (1997): 'Smooth non-parametric reciever operatin characteristic (ROC) curves for continuous diagnostic tests'. Statist. Med. 16, 2143-2156.

Zou KH, O'Malley AJ, Mauri L. (2007). Receiver-operating characteristic analysis for evaluating diagnostic tests and predictive models. Circulation, 6;115(5):654-7.

Impreso en papel bond 90 gr.
en familia tipográfica Candara a 11,5 pts.

Amadgraf Impresores Ltda.
Bogotá, D.C., Colombia
Octubre de 2020.

OTROS TÍTULOS DE ESTA COLECCIÓN

RADIACIÓN-MATERIA: GEANT4 Hands On!

GESTIÓN DE LA ENERGÍA: EL USUARIO DE ENERGÍA COMO PARTE ACTIVA DEL SISTEMA

GESTIÓN Y CIBERSEGURIDAD PARA MICRORREDES ELÉCTRICAS RESIDENCIALES

DETECCIÓN Y CORRECCIÓN DE PROPAGACIONES ANÓMALAS EN RADARES METEREOLÓGICOS

INTRODUCCION A LA INVESTIGACIÓN SOBRE DESASTRES NATURALES Y CIUDADES INTELIGENTES

INVESTIGACIÓN EN INGENIERÍA FUNDAMENTADA EN LA GESTIÓN DEL CONOCIMIENTO

LOS RECURSOS DISTRIBUIDOS DE BIOENERGÍAEN COLOMBIA

ARQUITECTURAS DE RED NEURO-CONVOLUCIONAL PARA APLICACIONES DE ROBÓTICA ASISTENCIAL

Vivimos en una era gobernada por datos, donde la intuición y el azar se han visto rezagados ante predicciones que soportan tanto decisiones cotidianas como grandes políticas gubernamentales. Una era donde la nueva riqueza se encuentra en los datos y en los métodos que permiten hacer un uso eficiente de éstos. En los últimos años, los métodos de procesamiento de datos que mayor precisión han presentado en tareas predictivas han sido aquellos basados en aprendizaje profundo, como por ejemplo las redes neuronales convolucionales. Este tipo de métodos representan un reto tanto por su alta exigencia de recurso computacional como por su complejidad de diseño e implementación.

Tomando como motivación lo anterior, en este libro el lector encontrará una guía práctica para la implementación, entrenamiento y validación de redes neuronales convolucionales usando Keras y acelerando con GPU. La guía se desarrolla mediante un caso de estudio típico enmarcado en las clasificaciones de imágenes satelitales. Adicionalmente la evaluación del modelo implementado incluye la comparación a nivel de speed-up con los modelos de redes neuronales pre-entenados más comunes: MobileNet, MobileNetV2, ResNet50 y VGG16.

ISBN 978-958-787-232-3

9 789587 872323