Detección y clasificación de células normales de la sangre periférica usando aprendizaje profundo

Wilson Alfredo Castro Zapata
Máster universitario de Bioinformática y Bioestadística
Área de Machine Learning

Nombre Consultor
Edwin Santiago Alférez Baquero

08/01/2020

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Detección y clasificación de células normales de la sangre periférica usando aprendizaje profundo.*
Nombre del autor:	*Wilson Alfredo Castro Zapata*
Nombre del consultor/a:	*Edwin Santiago Alférez Baquero*
Nombre del PRA:	
Fecha de entrega (mm/aaaa):	01/2020
Titulación:	*Máster universitario en Bioinformátia y Bioestadística*
Área del Trabajo Final:	*Machine Learning*
Idioma del trabajo:	*Español*
Palabras clave	*Aprendizaje Profundo, detección de objetos, clasificación de imágenes.*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

Mediante la utilización de herramientas de procesamiento digital de imágenes, se pueden extraer descriptores cuantitativos de las células en la sangre periférica, que a través de técnicas de Machine Learning (M.L) y Deep Learning (D.L.) permite diferenciar de forma automática y objetiva los tipos de células.

En este TFM se entrena mediante redes neuronales convolucionales (CNN) un sistema de detección y clasificación de células normales de la sangre que, al ingresar una imagen, detecta las células principales y luego permite recortar los leucocitos y clasificarlos según el tipo.

Para conseguir una mayor eficiencia y superar las dificultades que se presentan en las tareas de detección y clasificación, sobre todo en la determinación de descriptores que son realizados de forma artesanal, se aborda el problema con el uso de las arquitecturas de redes neuronales convolucionales con lo cual estos descriptores se obtienen de forma automática.

Para el desarrollo del clasificador se entrenó una red desde cero con un accuracy de 0.862 y un val_acuraccy de 0.858. También se entrenaron modelos como VGG16 y Mobile y se compararon los resultados.

Se emplearon en ambos entrenamientos la arquitectura de Keras con Tensorflow como backend con las últimas versiones.

Para el entrenamiento del detector se utilizó la arquitectura YOLO con la librería especializada ImageAI en detección de objetos y se ha obtenido un excelente desempeño.

Estos desarrollos tienen diversas aplicaciones en los campos de la bioinformática, y pueden dar las bases para que luego con la aplicación de metodologías similares a las propuestas en este TFM, se entrenen CNN para identificación de patologías a partir de la clasificación y detección de células anormales en la sangre.

Abstract

Through the use of digital image processing tools, quantitative descriptors of cells in peripheral blood can be extracted, which, through Machine Learning (M.L.) and Deep Learning (D.L.) techniques, allow cell types to be automatically and objectively differentiated.

In this TFM, a system of detection and classification of normal blood cells is trained through convolutional neural networks (CNN) which, when an image is entered, detects the main cells and then allows the leukocytes can be trimmed and classified according to the type.

To achieve greater efficiency and overcome the difficulties that arise in the tasks of detection and classification, especially in the determination of descriptors that are made by hand, the problem is addressed with the use of convolutional neural network architectures with the which these descriptors are obtained automatically.

For the development of the classifier a network was trained from scratch with an accuracy of 0.862 and a val_acuraccy of 0.858. Other models were reentrained with networks such as VGG16 and Mobile and the results were compared.

The latest versions of Keras architecture with Tensorflow as backend was used in both trainings.

For the training of the detector, the YOLO architecture was used with the specialized ImageAI library in object detection and excellent performance has been obtained.

These developments have various applications in the fields of bioinformatics, and might provide the basis for the application of methodologies similar to those proposed in this TFM, so CNN could be trained to identify pathologies from the classification and detection of abnormal blood cells.

Índice

Lista de figuras

1. Introducción.

1.1 Contexto Trabajo Final de Máster.

La sangre periférica (SP) es un fluido que circula a través de los vasos sanguíneos del cuerpo y transporta las células sanguíneas suspendidas en el plasma. Las células de la sangre se clasifican en leucocitos o glóbulos blancos (WBC del Ingles White Blood Cells), eritrocitos o glóbulos rojos (RBC del Ingles) y los trombocitos o plaquetas. Los leucocitos (WBC) se clasifican según la individualidad y las características de sus estructuras morfológicas, citoplasma y su núcleo (Young, 1972). Los glóbulos blancos o leucocitos se clasifican según sus características y funciones en dos linajes principales: el mieloide (neutrófilos, monocitos, eosinófilos y basófilos) y el linfoide (linfocitos T, linfocitos B y las células natural killer o células NK) (Orkin & Zon, 2008).
Los leucocitos son células con núcleo que tienen una función importante en la defensa contra las infecciones.

En el procesamiento de imágenes médicas, es un tema de gran importancia el análisis automático de imágenes microscópicas, y en este se han desarrollado numeroso estudios e investigaciones en la detección celular y en particular de las células en la sangre (Wang, Sun, & Yang, 2019). Sin embargo, debido a las complejidades inherentes en las células de la sangre como son los numerosos tipos de células con diferentes tamaños y formas, así como también a la adhesión entre las mismas, todo lo cual trae como resultado que la detección y localización de células de la sangre periférica con precisión sea una labor de mucha dificultad. Uno de los propósitos principales del examen microscópico de la sangre periférica, es que desempeña un papel importante en el campo del diagnóstico y control de las principales enfermedades.

1.2 Justificación del T.F.M.

El reconocimiento de las células de la sangre periférica se puede realizar de forma manual lo cual requiere que los especialistas, generalmente médicos, observen los frotis de sangre a través de microscopía óptica, utilizando su experiencia y pericia para discriminar y analizar las diferentes células, lo que requiere mucho tiempo, es un trabajo intensivo y tiene parte de subjetividad y por consiguiente tiene un margen de error por factores humanos. Por estas razones se ha elegido este tema sobre la detección y clasificación de células normales de la sangre periférica usando redes neuronales convolucionales, apoyándose a la vez en el enorme desarrollo de tecnologías y software (Kaiming, Xiangyu, & Shaoqing, 2018) que ha surgido en los últimos años para la realización de aplicaciones e investigación relacionada con el D.L. y que están generando un gran impacto en la forma de realizar estudios en el ámbito de la bioestadística y bioinformática y por supuesto en la aplicación práctica a nivel laboral.

Así mismo se espera dar un aporte con este TFM en el sentido de que la experiencia adquirida en este trabajo sirva de punto de referencia para otros estudios de Redes Neuronales Artificiales (CNN) para clasificar otras clases

más amplias de células, como por ejemplo células anormales relacionadas con enfermedades. Así mismo al abordar, explorar y aplicar las nuevas tecnologías y el software relacionado en el D.L. y en CNN a la aplicación de un problema específico, se pretende dar una memoria detallada del mismo que posibilite a otras personas interesadas en estas tecnologías y podrían provenir de muchos ámbitos académicos y profesionales dado el amplio espectro de aplicabilidad del D.L., que bien pueden tomar también como una guía y facilitar la comprensión de la forma de aplicación de estos recursos. Estos últimos puntos son importantes ya que al revisar la bibliografía sobre todo de artículos especializados, se acostumbra salvo algunas excepciones, a no mostrar el código base utilizado y los textos académicos quedan pronto obsoletos por la rápida aparición de actualizaciones o de nuevo software.

1.3 Problemática a resolver

En este TFM se propuso desarrollar un detector y clasificador de los diferentes tipos de células de la sangre con alta precisión utilizando CNN. Con este nuevo enfoque, no es necesario implementar la segmentación de imágenes, la extracción de características se vuelve automática y los modelos existentes se pueden ajustar para obtener clasificadores específicos (Rosebroock, 2017). Se analizaron imágenes de dataset públicos para el desarrollo de la metodología. Con los resultados obtenidos, será posible reentrenar las redes del Clasificador y el Detector obtenidos en este TFM con un dataset de imágenes propias de una institución hospitalaria y obtener bien sea una mayor precisión y generalidad o inclusive dar otro enfoque como la identificación de algunas patologías en la sangre.

De acuerdo con el dataset de las imágenes que se tuvieron disponibles para este desarrollo, fue necesario evaluar la arquitectura para el desarrollo de la red neuronal, entre YOLO que tiene un desarrollo más sencillo sobre FASTER R-CNN. Así mismo para el framework se revisaron posibilidades como fast.ai – pyTorch y Keras con Tensorflow como backend. La primera está más enfocada a la investigación que al desarrollo de aplicaciones, por lo cual finalmente se escogió a Keras – Tensorflow que adicionalmente se tiene las ventajas de la facilidad de uso por cuanto aplica los lenguajes de Python, R y Javascript entre otros que son de manejo del autor.

En el aspecto técnico y debido a los requerimientos de recursos inherentes a las redes neurales convolucionales, es necesario el uso de GPU y TPU. Todo el proceso se analizó que requeriría de un servidor en 4 CPU cores, unas 15GB de memoria RAM, 500 GB de disco duro y una Tesla K80 GPU.

El servidor gratuito de Google Colab junto a las capacidades de almacenamiento del Google Drive de la cuenta con la UOC fueron suficientes para este estudio. Se usó la Keras API para el backend de TensorFlow y el código escribió en Python con los notebooks de Jupyter.

1.4 Objetivos

1.4.1 Objetivo general:

- Desarrollar un detector y un clasificador de los diferentes tipos de células normales de la sangre periférica con alta precisión utilizando CNN.

1.4.2 Objetivos específicos:

- Adquirir los conocimientos de los modelos de D.L., especialmente los que usan Redes Neuronales Convolucionales CNN.
- Adquirir y cargar los conjuntos de datos de imágenes desde el disco, almacenarlos en la memoria y darles un formato de base de datos optimizado.
- Realizar el preprocesamiento de las imágenes de modo que sean adecuadas para la arquitectura de red escogida.
- Seleccionar e implementar la arquitectura popular de CNN que permita cumplir con la detección y clasificación de células de sangre normales.
- Evaluar la efectividad del modelo con métricas como el *validation accuracy,* que se obtiene con un conjunto de imágenes diferentes al utilizado en el entrenamiento (dataset de prueba o test).

1.5 Enfoque y metodología

Para realizar el presente trabajo final de máster, se tuvieron en cuenta los conocimientos y experiencia previos del autor, como se muestra brevemente en la siguiente tabla:

Áreas con experiencia previa	**Áreas sin experiencia previa**
- Conocimientos básicos de programación con Python del área cursada en el máster. - Conocimientos elementales en Machine Learning según se cursó en el máster. - Algoritmos básicos de clasificación.	- Redes neuronales convolucionales. - Arquitecturas y software de Deep Learning, en especial keras, tensorflow y YOLO. - Redes preentrenadas y su implementación como VGG16, Resnet, Inception, etc.

Metodología. El autor de acuerdo con las recomendaciones del tutor y según la disponibilidad y selección cuidadosa de imágenes realizada de plataformas públicas, se planteó utilizar para satisfacer los requerimientos de recursos inherentes a las redes neurales convolucionales, un servidor con GPU y TPU y una buena capacidad de memoria de almacenamiento de al menos 20GB.

El servidor de uso libre de Google Colab con la cuenta de Gmail y la capacidad de almacenamiento de Drive de la UOC satisfizo los requisitos planteados. No obstante, se consideró que todo el proceso pudiera requerir la configuración de un servidor ofrecido por empresas como la misma Google, IBM o Amazon, en especial el servicio Google Cloud Virtual Machine que da facilidades como un bono inicial de US$300. El servidor que se alquilara debería contar con una capacidad de 4 CPU cores, unas 15 GB de memoria RAM, 500 GB de disco

duro y una Tesla K80 GPU, así como tener en lo posible preinstalado el software necesario para la consecución de este proyecto como Python, Keras y Tensorflow entre otros.

Finalmente se obtuvieron buenos resultados con Google Colab en conjunto con las capacidades de almacenamiento del Google Drive de la cuenta con la UOC y no fue necesario alquilar un servidor pago. Se planteó desde el principio usar la Keras API para el backend de Tensorflow y el código y librerías necesarias en Python.

En este trabajo final de máster (TFM) se propuso:

1. Construir un **clasificador** con las imágenes etiquetadas para la clasificación automática entre cuatro tipos o grupos de células sanguíneas de los leucocitos o glóbulos blancos con alta precisión por medio de CNN. La técnica utilizada tradicionalmente se basa principalmente en la segmentación y extracción de características artesanales que luego se utilizan junto con herramientas de reconocimiento de patrones para la selección y clasificación de las diferentes células sanguíneas. Con el procedimiento propuesto mediante CNN, no se requiere implementar segmentación de imágenes, ya que con las CNN se realiza la extracción de características de forma automática y los modelos existentes que han sido desarrollados por científicos (LeCun & Bengio, 1995) y empresas de tecnología como Google y Microsoft entre otras, se pueden ajustar y entrenar con las imágenes de interés para obtener los clasificadores específicos.

Los diferentes tipos de células que se propuso clasificar automáticamente son los leucocitos correspondientes a los neutrófilos, eosinófilos, linfocitos y monocitos. Para esto se utilizaron dos datasets de imágenes de células normales que se combinaron en los conjuntos de entrenamiento y prueba/test.

2. Construir un **detector**-clasificador con una red YOLO utilizando el dataset con las etiquetas tanto de bounding boxes como de clasificación para diferenciar entre los tres principales grupos de células normales de la sangre, a saber, glóbulos rojos (RBCs), leucocitos o también conocidos como glóbulos blancos (WBCs) y plaquetas. Se utilizó un dataset de imágenes de células normales que se combinaron en los conjuntos de entrenamiento, prueba/test y validación.

3. Combinar los pasos 1 y 2, para realizar un **Detector** de los tres principales tipos de células sanguíneas y un **Clasificador** de los tipos principales de leucocitos.

En ambos pipelines se obtuvieron indicadores o métricas de desempeño como la exactitud o *accuracy* tanto con el conjunto de entrenamiento como el *val_accuracy* que evalúa la precisión en el conjunto de imágenes en el conjunto de prueba o test. Estas métricas son directamente proporcionadas por los algoritmos de entrenamiento y luego con los modelos resultantes se realizaron pruebas adicionales para la precisión. También se realizaron las gráficas de desempeño de estos indicadores durante el proceso de entrenamiento para el

Clasificador. Se guardaron los modelos obtenidos para que sean fácilmente vueltos a correr y se dispusieron en una cuenta pública de github con todos los archivos incluidos los notebooks para su consulta, revisión y reproducción por cualquiera que estuviera interesado.

1.6 Planificación del trabajo

Para la realización del presente TFM se definieron una serie de actividades o tareas a realizar para la consecución de los objetivos de este trabajo y se identificaron los logros e hitos que se debían obtener. Estos se especifican a continuación.

1.6.1 Actividades o tareas realizadas

- Documentación sobre el tema propio del TFM con textos especializados, cursos en plataformas como EDX y Coursera, artículos relacionados, otros trabajos finales de máster y tesis (del 2019/09/18 al 2019/12/31) para alrededor de 60 horas). Ya que se ha abordado una temática en la que las metodologías y el software se innovan continuamente, esto ha exigido la continua actualización por parte del autor.

- Aprendizaje y fundamentación del software requerido como Python y librerías específicas como fast.ai para clasificación de imágenes y tensorflow y arquitecturas como keras y las redes neurales convolucionales (del 2019/09/18 al 2020/12/31 para alrededor de 80 horas). Ya que se ha abordado una temática en la que las metodologías y el software se innovan continuamente, esto ha exigido la continua actualización y consulta de información por parte del autor.

- Búsqueda del dataset de imágenes apropiadas (10 horas) y verificar la factibilidad técnica y de derechos de autor para usarlo en el entrenamiento del modelo.

- Balancear el dataset con técnicas como submuestreo, repetición, recorte, edición con procesos como rotación. En este caso fue importante que el balanceo no implique un crecimiento artificial de alto impacto en el dataset que pueda llevar a que el estudio realizado no sea confiable (del 2019/10/05 al 2019/10/20 para 15 horas).

- Implementaciones de prueba con servidor de Google Colab y definir si se pueden cumplir los requerimientos técnicos para cumplir con los objetivos del TFM. Si se descartaba el servidor de Google colab, se debía alquilar y configurar un servidor con el software requerido (del 2019/10/21 al 2019/10/28 para unas 20 horas). En este caso se obtuvieron buenos resultados con Google Colab y no fue necesario alquilar un servidor.

- Carga del dataset al servidor y pruebas preliminares de la metodología a aplicar sobre las imágenes. Se revisó la posibilidad de aplicar las metodologías

propuestas por varios autores como Chollet y Rosebrook, así como otras que estuvieron disponibles producto de artículos de investigación y TFM (del 2019/10/29 al 2019/11/03 para aproximadamente 30 horas).

- Realizar el entrenamiento de la red: aplicar procedimientos de submuestreo del dataset de forma aleatoria y balanceada de las imágenes organizándolas en tres grupos: training, validation y test (del 2019/11/03 al 2019/11/18 para 30 horas).

- Realimentación y mejora del modelo, adaptándolo a los datos y ajustando los parámetros e hiperparámetros requeridos (15 horas).

- Llevar a cabo el proceso de training, validación y prueba con la metodología y arquitectura aplicadas (20 horas).

- Analizar los resultados finales (10 horas).

- Redacción del producto final (30 horas hasta el 8 de enero para su entrega).

- Grabación del video con la presentación y sustentación del TFM (10 horas hasta el 12 de enero)

1.6.2 Planificación con temporización y Diagrama de Gantt.

Se elabora el diagrama de Gantt para estas actividades para lo cual se tienen en cuenta las ajustadas fechas para las PECs.

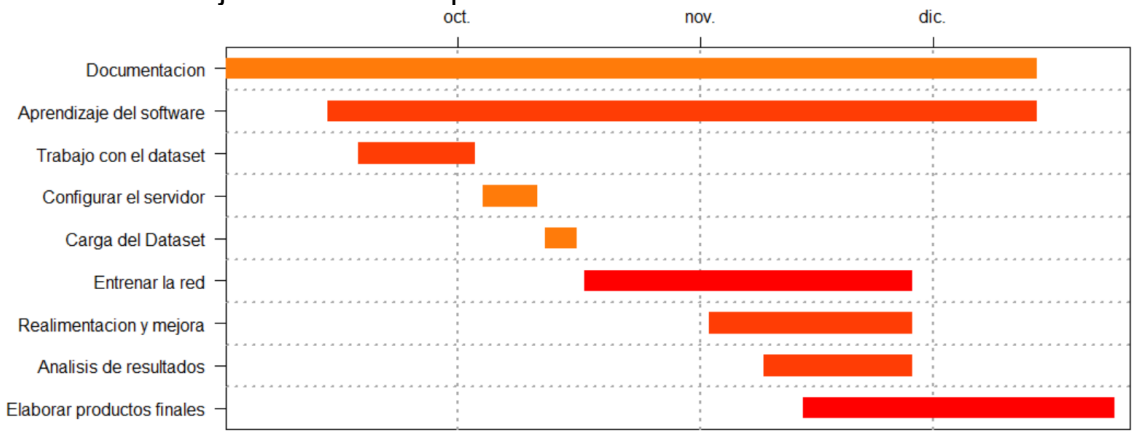


Figura 1. Diagrama de Gantt para la temporización de las actividades de este T.F.M. Fuente: El autor.

1.6.3 Hitos

- Obtener el dataset apropiado y balancearlo.
- Configuración del servidor.
- Cargar el dataset en el servidor
- Realizar el entrenamiento del modelo Clasificador con D.L..
- Realizar el entrenamiento del modelo Detector con Deep Learning.

- Combinar el Detector y el Clasificador y realizar la corrida total de estos componentes con varias imágenes y establecer la precisión obtenida.
- Redactar la memoria del TFM.
- Elaborar la presentación en video del TFM.

1.6.4 Riesgos asociados al TFM.

1. No cumplir con alguna de las actividades según el cronograma, lo cual puede ser de alto riesgo para la consecución satisfactoria del plan de trabajo. El riesgo es un poco mayor en los meses de octubre y parte de noviembre, pero dado que el autor es docente y finaliza sus actividades laborales en la primera semana de diciembre, tendrá la posibilidad de dedicación a tiempo completo en ese último mes para ponerse al día si fuera necesario.

2. Que se presente alguna contingencia externa al proyecto que pueda afectar al autor en el desempeño. Esto es imposible de preveer y solo cabe esperar que no se vaya a presentar.

3. Riesgo de que se presenten problemas con el dataset, dado que normalmente las imágenes no son del mismo tamaño o están muy desbalanceadas y ello puede generar problemas al alimentarlas al modelo. Para minimizar este riesgo se hará una selección cuidadosa del dataset y se utilizan las técnicas apropiadas para ajustar las imágenes así como balancear el dataset de estas para que haya proporcionalidad entre las diferentes categorías.

4. Cambios en las tecnologías usadas. Ya que se hace uso de las arquitecturas de mayor novedad y de continuo desarrollo en los últimos años como tensorflow que ahora incorpora keras, se pueden presentar incompatibilidades con otras versiones y puede afectar algún código o modelo implementado. Además, al hacer uso de la plataforma de Google Colab esta advierte que dejará de dar soporte a algunas versiones anteriores.

5. sobrecostos del servidor. Si bien se pretende inicialmente utilizar un servidor de Google y esta empresa otorga $300 lo cual podría eventualmente ser suficiente para el desarrollo, también podría ocurrir que se sobrepase ampliamente este costo. Para minimizar este riesgo, el autor usará en principio el servicio gratuito de Google llamado Colab en la fase de aprendizaje del autor y luego será cuidadoso con el tiempo de alquiler del servidor.

6. Riesgo con los archivos resultantes del proyecto por pérdida o daño. Se minimiza este riesgo teniendo los archivos en servicios en la nube como Google drive y realizando backups de los archivos que se van obteniendo y colocándolos también en Google drive.

1.6.5 Costos asociados al desarrollo del proyecto.

Fueron básicamente:

- Los correspondientes a costo de energía en el domicilio del autor (alrededor de 40€).

- Costo de oportunidad implícita por la dedicación al proyecto por parte del autor el cual es consciente que se trata más bien de una inversión a mediano plazo. Se menciona este asunto, pero por estas mismas razones no se considera objetivo fijar un valor.

- Costo de asesoría correspondiente en este caso al tutor de la UOC, que si bien ya está cubierto por el valor de la matrícula del TFM, ha de ser alto por tratarse de personal altamente calificado. Se puede considerar que corresponde al 30% de la matrícula, es decir, 0.3*1.307,09 €= 392.127€

- Costo de alquiler del servidor. Se planteó desde la estructuración de este TFM que inicialmente se usará en la fase de aprendizaje por parte del autor el servidor de Google en el servicio Google - Colab que es libre de recargo. Con este servidor y con la capacidad de almacenamiento del Drive de la cuenta Gmail de la UOC, se tuvieron los recursos necesarios y no se incurrió en costos de alquiler de otro servidor.

-El software utilizado es básicamente de acceso libre y no presenta costos. El software como editores de texto y edición de video ya los tenía el autor, por lo cual no implica costos en el desarrollo de este TFM.

-Costos relacionados con los productos finales como el video de la presentación, la diagramación del documento final. Estos los realizó el autor por lo que no se incurrió en costos adicionales. El equipo de grabación también ya lo poseía el autor. No obstante, este trabajo así sea por al autor se debe considerar y se valoran en 300€.

Costo total del TFM: 732.13€

1.6.6. Breve sumario de productos obtenidos

Se realizará y entregará los siguientes productos al finalizar el trabajo final del máster:

- Plan de trabajo.
- Memoria.
- El software desarrollado con el código disponible de forma pública en Github (https://github.com/alfredwilson/TFM, s.f.).
- Un video con la presentación y sustentación del TFM.
- Una autoevaluación del proyecto.

Todos estos entregables se elaboraron de acuerdo con las indicaciones y plantillas referidas en el aula de la UOC.

1.6.7 Descripción de los siguientes capítulos.

En los siguientes capítulos se hará una contextualización de los temas relacionados con este proyecto, iniciando con el aprendizaje automático (Machine Learning M.L.), el aprendizaje profundo (Deep Learning D.L.) y las redes neuronales convolucionales (CNN por sus siglas en Inglés de Convolutional Neural Networks). Se abordarán tecnologías y metodologías del estado del arte en reconocimiento de imágenes y el software específico, en especial las librerías Keras y Tensorflow como backend para el desarrollo de este TFM.

También se incluirán los materiales y los métodos, el procedimiento usado en la selección del dataset y se realizará una descripción de cómo está compuesto el mismo y quién lo liberó y el proceso para su balanceo y carga en el servidor.
Se describirá cómo se va a proceder con el submuestreo del dataset en tres grupos: *training, validation* y *test*. Luego se describirán los resultados obtenidos en los capítulos finales y como se ajustaron los parámetros e hiperparámetros para llegar a obtener buenos resultados. Para finalizar la memoria del TFM se presentarán los anexos y la bibliografía.

2. Contexto y antecedentes bibliográficos

En esta sección brevemente se abordan las temáticas relacionadas con la Visión por Computador que es una de las áreas que ha avanzado rápidamente gracias al aprendizaje profundo o DL. Como estas temáticas son muy extensas, en muchos casos solo se mencionarán los conceptos o se darán algunas definiciones y explicaciones básicas.

2.1. Inteligencia artificial, Machine Learning y Deep Learning

Se precisa aquí que la terminología que se va a usar en este TFM va a incluir algunos términos del Inglés por ser muy aceptados también en textos y artículos en español como son Machine Learning (M.L.) para el Aprendizaje Automático, Deep Learning (D.L.) para Aprendizaje Profundo y CNN para Redes Neuronales Convolucionales.

En los primeros enfoques de la Inteligencia Artificial (I.A.), se intentaron programar explícitamente el conocimiento necesario para realizar determinadas tareas; sin embargo, estos enfrentaron dificultades para tratar problemas complejos del mundo real como en la clasificación de imágenes, ya que resulta un trabajo en extremo exigente el diseño manual por parte de los ingenieros de todos los detalles necesarios para que un sistema de I.A. logre los resultados que se esperan (Rosebroock, 2017).

El campo del M.L. ha progresado enormemente al abordar tareas complejas como los tratados en este TFM que son la de clasificación de imágenes y la detección de objetos en una imagen.

El M.L. proporcionó soluciones más viables con la capacidad de mejorar a través de la experiencia y los datos. El M.L. se puede definir como el proceso de inducir inteligencia en un sistema o máquina sin programación explícita (Andrew NG, profesor adjunto de la Universidad de Stanford).

Aunque el M.L. puede extraer patrones de los datos, existen limitaciones en el procesamiento de los datos originales que depende en gran medida de las características diseñadas artesanalmente, por lo cual ha resultado ser muy eficaz el llamado D.L. que realiza este proceso automáticamente. El D.L. es una subcategoría del M.L. en la inteligencia artificial (IA) que trata con algoritmos inspirados en la arquitectura biológica y el funcionamiento del cerebro para ayudar a las máquinas con inteligencia.

En la figura 2 se observa la relación entre estos esquemas de la Inteligencia Artificial.

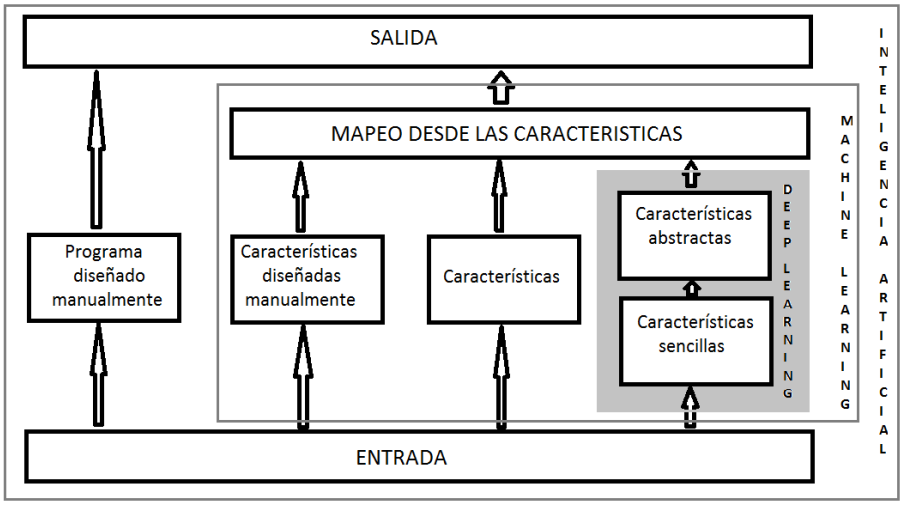


Figura 2. Relaciones y esquemas de alto nivel de inteligencia artificial (A.I.), Machine Learning y Deep Learning. Fuente: el autor.

Para cada imagen del conjunto de datos, se realiza la extracción de características, o el proceso de tomar una imagen de entrada, cuantificándola de acuerdo con algún algoritmo llamado extractor de características o descriptor de imagen, y devolviendo un vector o lista de números que pretende cuantificar el contenido de una imagen como forma, textura, color, etc. Estos vectores de características servían como entradas para los modelos de M.L. (Rosebroock, 2017).
La Figura 3 muestra el proceso de cuantificación de una imagen que contiene células de leucocitos a través de una serie de descriptores de color, textura y forma de la célula.

Figura 3. Cuantificación de los contenidos de imágenes de leucocitos a través de Descriptores de Imagen de tipo Color Textura y Forma. Fuente: el autor.

El D.L. y específicamente las CNN, adoptan un enfoque diferente. En lugar de definir manualmente un conjunto de reglas y algoritmos para extraer características de una imagen, estas características se aprenden automáticamente del proceso de aprendizaje.

El D.L. comenzó en 1958 cuando el psicólogo de Cornell, Frank Rosenblatt, desvela el Perceptrón, una computadora del tamaño de una habitación

consistente de una red neuronal de una sola capa. Luego en 1969 el gigante de la inteligencia artificial, Marvin Minsky del MIT, escribe un libro arrojando dudas sobre la viabilidad de las redes neuronales, lo cual retrasó y casi elimina el desarrollo de estas. En 1986, el pionero de las Redes neuronales, Geoffrey Hinton y otros, encuentran una manera de entrenar redes neuronales multicapa para corregir errores y se produce entonces mucha actividad relacionada con este tipo de redes. En 1989 el investigador francés Yann LeCun y luego en laboratorios Bell, comienza el trabajo fundacional en un tipo de red neuronal que se vuelve crucial para el reconocimiento de imágenes (Lecun, 1986).

2.2 Redes Neuronales Artificiales (CNN) y el Perceptrón

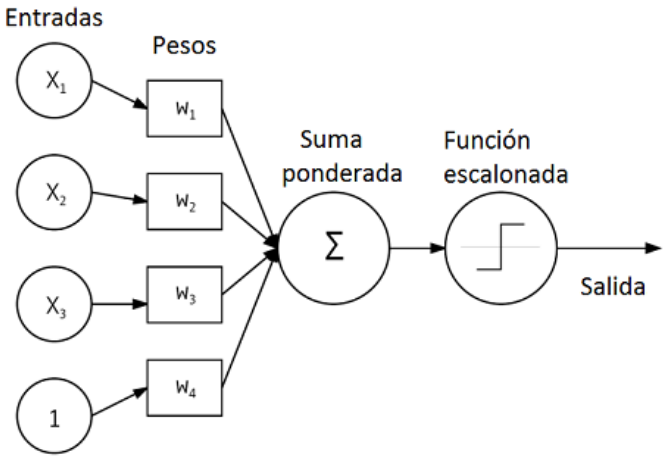


Figura 4. Ejemplo de una arquitectura de red simple Perceptrón que acepta una cantidad de entradas, con unos pesos calcula una suma ponderada y aplica una Función escalonada para obtener la predicción final o salida.

Fuente(https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53)

Entrada de las redes neuronales artificiales: la neurona artificial toma una entrada que, en función de los pesos de las conexiones, puede ignorarse (por un peso de 0.0 en una conexión de entrada) o pasar a la función de activación.

La función de activación también tiene la capacidad de filtrar datos si no proporciona un valor de activación distinto de cero como salida (Figura 4).

Se expresa la entrada neta a una neurona como los pesos en las conexiones multiplicadas por la activación entrante en la conexión, como se muestra en la Figura 4. Para la capa de entrada, solo se toma la entidad en ese índice específico, y la función de activación es lineal (pasa el valor de la entidad). Para las capas ocultas, la entrada es la activación de otras neuronas.

Valores de ponderación (pesos) de conexión: Los valores de ponderación en las conexiones en una red neuronal son coeficientes que escalan (amplifican o minimizan) la señal de entrada a una neurona determinada en la red. A menudo, las conexiones se anotan como *w* en representaciones matemáticas de redes neuronales.

Los Sesgos: son valores escalares agregados a la entrada para asegurar que al menos unos pocos nodos por capa se activen independientemente de la intensidad de la señal. Los sesgos permiten que el aprendizaje suceda al dar acción a la red en caso de baja señal. Permiten a la red probar nuevas interpretaciones o comportamientos. Los sesgos generalmente se denotan por *b*, y, al igual que los pesos, los sesgos se modifican a lo largo del proceso de aprendizaje.

Funciones de activación: Las funciones que gobiernan el comportamiento de la neurona artificial se llaman funciones de activación. La transmisión de esa entrada se conoce como propagación hacia adelante. Las funciones de activación transforman la combinación de entradas, pesos y sesgos. Los productos de estas transformaciones se ingresan para la siguiente capa de nodo. Muchas (pero no todas) las transformaciones no lineales usadas en redes neuronales transforman los datos en un rango conveniente, como 0 a 1 o -1 a 1. Cuando una neurona artificial pasa un valor distinto de cero a otra neurona artificial, se dice que es activado.

El D.L. puede aprender características complejas al combinar características más simples aprendidas de los datos de múltiples capas no lineales, denominadas arquitecturas de D.L., con las que se pueden descubrir representaciones jerárquicas de datos con unos niveles de abstracción crecientes (Moolayil, 2019).

Como resultado, las CNN aprenden a responder a las diferentes características de la imagen (bordes, formas, etc.) como filtros en vez de los algoritmos tradicionales de M.L.. En la figura 5 se ilustra este proceso en la clasificación de imágenes.

En el D.L., se trata de abordar el problema en términos de una jerarquía de conceptos en que cada uno de estos se construye sobre los demás. Los conceptos en las capas de nivel inferior de la red codifican alguna representación básica del problema, mientras que las capas de nivel superior usan estas capas básicas para formar conceptos más abstractos. Este aprendizaje jerárquico permite eliminar por completo el diseño artesanal y tratan a las CNN como herramientas de aprendizaje en todo el proceso.

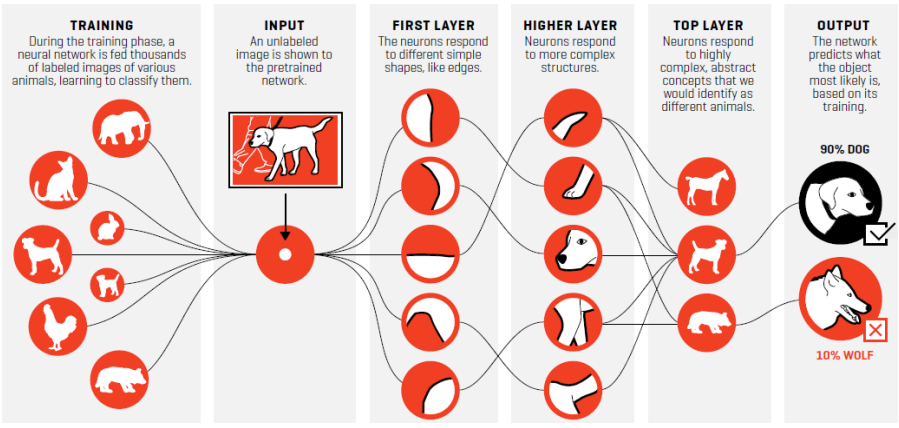


Figura 5. D.L. para reconocimiento de imágenes

Fuente: https://fortune.com/longform/ai-artificial-intelligence-deep-machine-learning/

La tecnología y el conocimiento actual han hecho cambiar las cosas y resurgir el D.L., de tal manera que ahora se tiene:
1. Computadoras más rápidas
2. Hardware altamente optimizado como GPU y TPU.
3. Grandes conjuntos disponibles de datos etiquetados del orden de millones de imágenes.
4. Una mejor comprensión de las funciones de inicialización de los pesos o ponderaciones y lo que funciona o no funciona.
5. Funciones de activación superiores.

El modelo de D.L. que se va a trabajar en este TFM es la categoría llamada Aprendizaje Supervisado, en el cual se hace un entrenamiento con imágenes que están etiquetadas. Teniendo en cuenta los datos de entrenamiento, un modelo tipo "Clasificador" se crea a través de un proceso de entrenamiento donde se hacen predicciones sobre los datos de entrada (imágenes) y luego se corrigen cuando las predicciones son erróneas al comparar estos resultados con la etiqueta correspondiente al dato de entrada (imagen). Este proceso de entrenamiento continúa hasta que el modelo logra algún criterio de parada deseada, tal como una baja tasa de error (loss) o un número máximo de iteraciones de entrenamiento (epoch).

En el contexto de Clasificación de Imágenes el conjunto de datos de imagen se compone de las correspondientes imágenes junto con su etiqueta de clase que se pueden utilizar para enseñar al clasificador de aprendizaje automático lo que cada categoría "parece ser". Si el clasificador hace una predicción incorrecta, entonces se pueden aplicar métodos para corregir el error.

Se observa en la siguiente figura un esquema elemental de una red neuronal

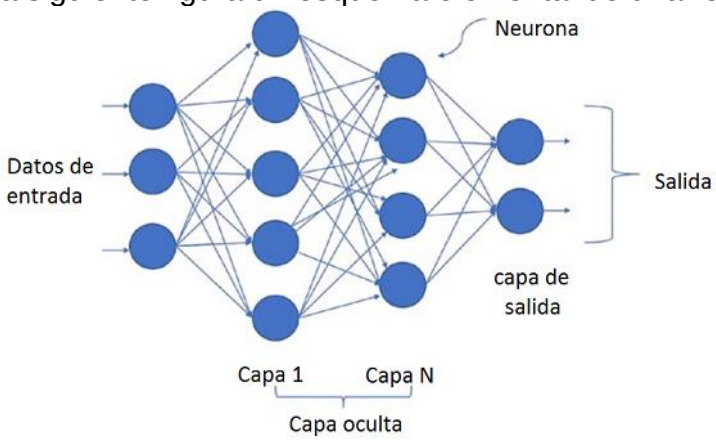


Figura 6. Esquema elemental de una red neuronal con datos de entrada, una capa oculta y una capa de salida. Fuente: (Moolayil, 2019)

De acuerdo con la figura 6, los datos de entrada son tomados por las neuronas en la primera capa oculta, que luego proporciona una salida a la siguiente capa y así sucesivamente, lo que finalmente resulta en la salida. Cada capa puede tener una o muchas neuronas, y cada una de ellas calculará una función como la de activación. La conexión entre dos neuronas de capas sucesivas tendría un peso asociado. El peso define la influencia de la entrada a la salida para la

próxima neurona y, finalmente, para la salida final general. En una red neuronal, todos los pesos iniciales son aleatorios en el proceso de entrenamiento del modelo, estos pesos se actualizan iterativamente para aprender a predecir una salida correcta. Al descomponer la red, se observan r algunos bloques de construcción lógicos como neurona, capa, peso, entrada, salida, una función de activación dentro de la neurona para calcular un proceso de aprendizaje, y así sucesivamente (Moolayil, 2019).

Las arquitecturas del D.L. son:
- Redes Neuronales Profundas (Deep Neural Networks).
- Redes Neuronales Convolucionales (CNN)
- Redes Neuronales Recurrentes (Recurrent Neural Networks RNN).
- Arquitecturas emergentes.

De todas estas sólo se abordan en este TFM las DNNs, las CNN y se mencionarán algunas de las arquitecturas emergentes.

-Deep Neural Networks (DNN). La estructura básica de las redes neuronales profundas (DNN) consiste en una capa de entrada, varias capas ocultas y una capa de salida (Figura 6). Una vez que se proporcionan los datos de entrada a las DNN, los valores de salida se calculan secuencialmente a lo largo de las capas de la red. En cada capa, el vector de entrada que comprende los valores de salida de cada unidad en la capa inferior se multiplica por el vector de ponderación para cada unidad en la capa correspondiente para producir la suma ponderada. Luego, se aplica a la suma ponderada una función no lineal, como la logística, tangente hiperbólica o la Unidad Lineal Rectificada (ReLU), para calcular los valores de salida de la capa. El cálculo en cada capa transforma las representaciones en la capa inferior en representaciones ligeramente más abstractas.

- CNN son arquitecturas que han tenido éxito particularmente en el reconocimiento de imágenes y consisten en capas de convolución, capas no lineales y capas de agrupación. La representación de la imagen de una CNN en diferentes capas es similar a cómo el cerebro procesa la información visual.

Uno de los desafíos que tienen los problemas de visión por computadora es que las entradas pueden ser realmente grandes. Por ejemplo, con imágenes de 64 por 64 pixeles que al tomar la profundidad dada por el color se vuelve 64 por 64 por 3 porque hay tres canales de color y al multiplicar el resultado es 12288. Entonces el vector de características de entrada tiene una dimensión de 12288 para esta imagen que es en realidad muy pequeña.

Al trabajar con imágenes más grandes, como 1000 x 1000 píxeles que es solo un megapíxel y con los tres canales RGB, el tamaño es de tres millones. Si en la primera capa oculta se tienen 1000 unidades ocultas, luego el número total de pesos al usar una red estándar o totalmente conectada, esta matriz tendría una dimensión de 1000 por 3 millones, es decir, tres mil millones de parámetros que es tremendamente grande. Con tantos parámetros, es difícil obtener suficientes datos para evitar que una red neuronal se sobreajuste. También

podrían no ser viables los requisitos computacionales y de memoria para entrenar una red neuronal con tres mil millones de parámetros.

Para lograr el uso de imágenes grandes, se necesita implementar mejor la operación de convolución que es uno de los pilares fundamentales de las CNN.

2.3 La operación convolucional.

La operación convolucional es uno de los bloques de construcción fundamentales de una CNN. Uno de sus usos primordiales en el proceso de reconocimiento de objetos en imágenes es que las primeras capas realizan la detección de bordes horizontales y verticales y luego algunas capas posteriores podrían detectar partes de objetos completos.

La convolución es una operación matemática sobre dos funciones para producir una tercera función que expresa cómo la forma de una es modificada por la otra. El término convolución se refiere tanto a la función del resultado como al proceso de cálculo. En una red neuronal, se realiza la operación de convolución en la matriz de imagen de entrada para reducir su forma. En el siguiente ejemplo, se realiza la convolución de una imagen en escala de grises de 6 x 6 con una matriz de 3 x 3 llamada filtro o núcleo para producir una matriz de 4 x 4.

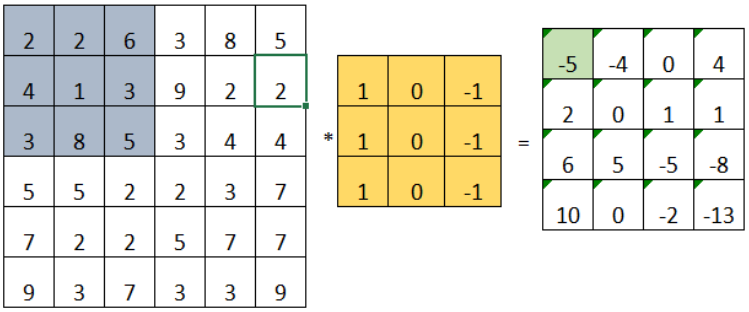


Figura 7. Ejemplo de la operación de convolución con una entrada de 6x6 y un filtro de 3x3
Fuente: el autor.

En la figura 7 se tiene de entrada una imagen de 6 x 6 en escala de grises, luego es sólo una matriz de 6 x 6 x 1, en lugar de 6 x 6 x 3 porque no hay canales RGB separados.

Con el fin de detectar los bordes, como los bordes verticales en esta imagen, se puede construir una matriz de 3 x 3 que en la terminología de las CNN es el llamado **filtro**. El filtro para reconocer bordes verticales es como se muestra en la figura, que es una matriz 3 x 3 con las columnas [1 1 1, 0 0 0, -1 -1 -1]. La suma de todos los elementos de un filtro normalmente es cero y el tamaño es cuadrado con dimensiones impares, los más comunes son de 3x3 y 5x5. Algunos trabajos de investigación llaman a esta matriz un núcleo en vez de un filtro.

Luego se toma la imagen de 6 x 6 y se le aplica la convolución que se denota con el asterisco. Algo desafortunado acerca de la notación se presenta porque el asterisco es el símbolo estándar para la convolución en matemáticas, pero en Python el asterisco también se utiliza para denotar la multiplicación elemento a elemento entre dos matrices. Así que este asterisco tiene dos propósitos, lo cual se conoce como notación sobrecargada.

El resultado de este operador de convolución es una matriz de 4 x 4, que se puede interpretar e imaginar como una imagen de 4 x 4.

Para calcular el primer elemento de la matriz de 4x4, se puede visualizar al tomar el filtro de 3 x 3 (en naranja) y se pega en la parte superior de la región de 3 x 3 de la imagen de entrada original, es decir la matriz de 6x6 (zona azul).

Primero, se realiza el producto punto entre el filtro y los primeros 9 elementos de la matriz de imagen y se obtiene el primer elemento la matriz de salida. Esto se realiza con la multiplicación elemento a elemento entre los elementos de la matriz del filtro y los correspondientes 9 primeros elementos de la matriz de 6x6 y luego se realiza la suma de estos productos (producto punto o escalar), según:

(2*1) + (2*0) + (6*(-1)) + (4*1) + (1*0) + (3* (-1)) +(3*1) + (8*0) + (5*(-1)) = -5

A continuación, para calcular el segundo elemento de la matriz de 4x4, se toma el filtro sobre la imagen de entrada (representado por el cuadrado azul) y se desplaza un paso a la derecha. Luego se realiza la misma operación del producto escalar. Aquí el autor realizó una pequeña simulación en Excel, donde se aplicaron las funciones de SUMAPRODUCTO() con procedimientos típicos en Excel. En los programas y API típicos de D.L. se implementan funciones específicas para la operación de convolución, así en Python se usa la función llamada "conv_forward" y con Tensorflow la función a aplicar es tf.nn.conv2d. Otros frameworks de D.L. como Keras, hay una función llamada Conv2D que implementa la convolución, pero todos los frameworks de D.L. que tienen un buen apoyo para convolución, tendrán alguna función para la implementación de este operador.

Este procedimiento como se ha explicado para obtener el primer elemento de la matriz (imagen) de salida, continúa hasta barrer toda la imagen de 6x6 con el filtro de 3x3, desplazando este filtro sobre la imagen de entrada (matriz 6x6 en el ejemplo dado) hacia la derecha una unidad y al llegar al extremo derecho entonces una fila hacia abajo, con lo cual se obtiene la matriz de 4x4 de la derecha como resultado de la convolución. El paso o stride en este caso es de 1, aunque se puede utilizar un paso de diferente valor.

En resumen, la matriz de la izquierda es conveniente interpretarla como una imagen, la que está en el medio, se la interpreta como un filtro. La de la derecha, se puede interpretar como otra imagen. Todo el proceso es un detector de bordes vertical.

En la figura 8 se observa una imagen en la cual es más oscuro a la izquierda y más brillante a la derecha. Esto se representa con valores de cero a la izquierda y de valores de 10 en la mitad derecha de la imagen. Al realizar la convolución con el filtro de detección de bordes, se termina con 30 negativos en el centro de la imagen resultante. En la figura 8 se observa en la parte inferior las franjas claras y oscuras de las imágenes de entrada y la de salida correspondiente.

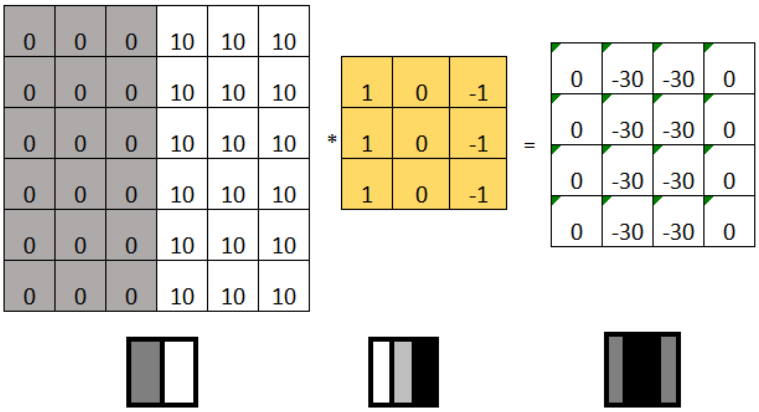


Figura 8. Detección de franjas verticales con franjas claras y oscuras con el tipo de filtro para este fin. Fuente (Andrew, 2019)

2.3.1 El Padding o relleno

Con el fin de construir redes neuronales profundas una modificación para la operación convolucional básica que se necesita utilizar es el padding o relleno.

En la situación anterior se tiene una imagen de seis por seis y al hacer la convolución con un filtro de tres por tres, se termina con una salida de cuatro por cuatro como una matriz de cuatro por cuatro. Las matemáticas precisan así que una imagen *n x n* al hacer la convolución con un filtro de tamaño *f x f*, entonces la dimensión de la salida será:

$$Matriz\ salida = M_{(n-f+1)x(n-f+1)}$$

Que para el caso de la figura 9 es

$$Matriz\ salida = M_{(6-3+1)x(6-3+1)} = M_{4x4}$$

Se observa entonces que la imagen resultante (por la matriz de salida) luego de aplicar una convolución es más pequeña lo cual es una desventaja, ya que al aplicar varias convoluciones se puede llegar a tener imágenes resultantes muy pequeñas. Una forma de disminuir ese “encogimiento” acelerado de las imágenes por las convoluciones es rellenando la imagen a la que se le va a aplicar la convolución con uno o varios bordes adicionales de 1 pixel cada uno alrededor de a imagen. Este es el relleno o padding. Con un padding de 1 para la imagen de 6x6, esta queda de 8x8. En este caso con padding p, la matriz resultante ya que la imagen se incrementa en 2p, es (Andrew, 2019):

$$Matriz\ salida = M_{(n+2p-f+1)x(n+2p-f+1)}$$

Para una imagen de tamaño 6x6 (n=6), con un filtro de tamaño 3x3 (f=3) y con un padding p=1, se tiene una imagen de salida de

$$Matriz\ salida = M_{(6+2*1-3+1)x(6+2*1-3+1)} = M_{6x6}$$

Luego la imagen resultante no se encoje. Por ejemplo, la figura 9 es representativa de este caso, en que se observa el borde de ceros alrededor de la matriz correspondiente a la imagen original de 6x6, y con esto queda de 8x8:

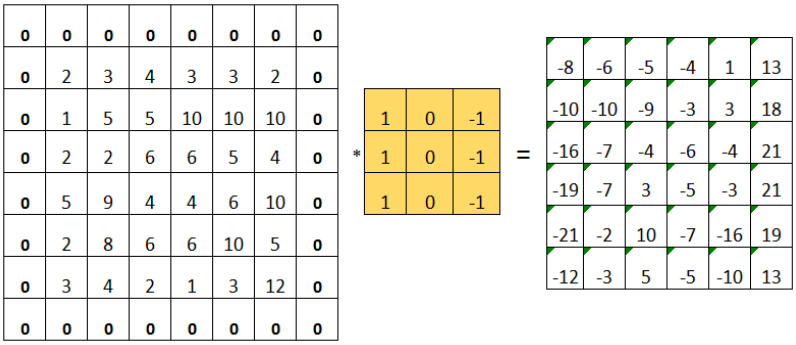


Figura 9. Imagen de tamaño 6x6 con un padding o relleno de 1. Al realizar la convolución con un filtro de tamaño 3x3 se obtiene una imagen del mismo tamaño que la de entrada. Fuente el autor.

Cuando el paso o stride (s) no es de la unidad, la matriz resultante (o imagen de salida), es de tamaño:

$$Matriz\ salida = M_{\left(\frac{n+2p-f}{s}+1\right)x\left(\frac{n+2p-f}{s}+1\right)}$$

Así, con una imagen de entrada de 7x7 y un filtro de 3x3, con stride s=2 y padding p=1, se tiene también que n=7 y f=3, se tiene una matriz de salida de tamaño:

$$Matriz\ salida = M_{\left(\frac{7+2*1-3}{2}+1\right)x\left(\frac{n+2p-f}{s}+1\right)} = M_{4x4}$$

Y ya no se podría ingresar una imagen de 6x6 con estos parámetros porque no da cantidades enteras, en un caso de estos, algunos algoritmos redondean. Todos estos parámetros como son el tamaño *n x n* de la imagen de entrada, el filtro de tamaño *f x f* a aplicar, el paso o stride y el relleno o padding se deben definir bien al diseñar una red o al utilizar una preentrenada.

2.3.2 Convolución sobre volúmenes

Se han tratado hasta aquí las convoluciones sobre las imágenes 2D, que son las correspondientes a imágenes en una escala de grises. Cuando se tiene una imagen a color como RGB, se agrega a la dimensión de la imagen los 3

canales del color. Así una imagen de color de tamaño *nxn* da lugar a un volumen de *nxnx3*, con el tres que corresponde a los tres canales de color.

En la figura 10 se tiene el caso de una imagen RGB de 6x6, que es como una pila de tres imágenes de seis por seis. Con el fin de detectar bordes o alguna otra característica en esta imagen, se puede realizar la convolución con un filtro 3D de 3x3x3. Así que el filtro también tendrá tres capas correspondientes a los canales rojo, verde y azul.

En la imagen de entrada, los primeros 6 corresponden a la altura de la imagen, los siguientes 6 al ancho, y el 3 es el número de canales. El filtro también tiene una altura, un ancho y un número de canales. El número de canales en la imagen de entrada debe coincidir con el número de canales en el filtro a utilizar. La salida de la operación de convolución será una imagen de 4x4x1, es decir, que se vuelve 2D.

Para realizar la convolución, el filtro es de 3x3x3 o sea de 27 números o parámetros. Luego cada uno de estos 27 números se multiplican con los números correspondientes en los canales rojo, verde y azul de la imagen, así que toma los primeros nueve números del canal rojo, luego los que están debajo de este que son el canal verde y luego los del canal azul y se multiplican con los 27 números correspondientes al filtro. A continuación se suman todos esos productos con lo que se obtiene el primer número en la salida.

Para calcular la siguiente salida se toma el filtro y se desliza una unidad y de nuevo se realizan las 27 multiplicaciones y se suman los 27 productos resultantes con lo cual se obtiene la próxima salida y así sucesivamente.

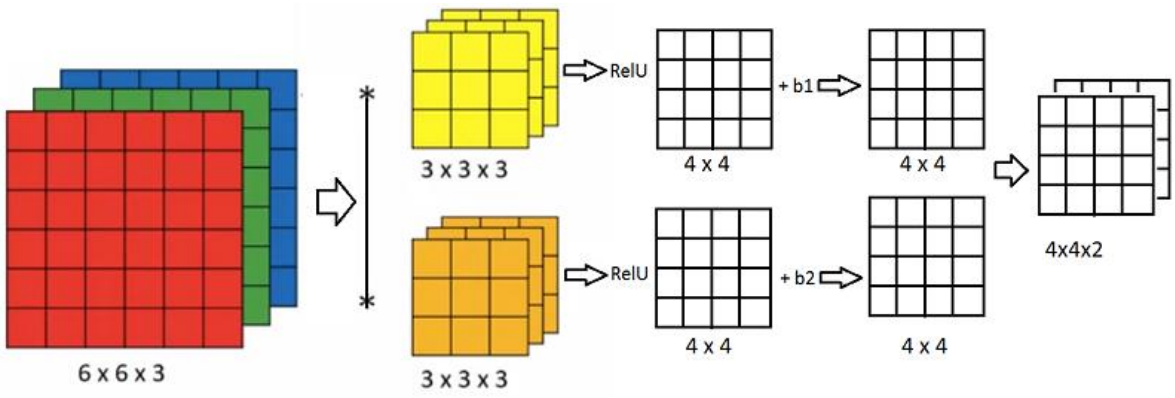


Figura 10. Convolución en imágenes a color RGB con dos filtros. (Andrew, 2019)

También se pueden aplicar varios filtros a la vez, por ejemplo uno para detectar bordes verticales y otro filtro para detectar bordes horizontales, lo cual es fundamental para construir CNN. En la figura 10 se ilustra el caso de una imagen de entrada de 6x6x3 con un fitro de 3x3x3 que al realizar la convolución produce una salida de 4x4 que podría ser para aprender a detectar un borde vertical o aprender a detectar alguna otra característica. Luego se tiene un segundo filtro que bien podría ser un detector de bordes horizontales. Al

realizar la convolución con el primer filtro se produce una primera salida de tamaño 4x4 y la convolución con el segundo filtro da otra salida diferente pero también de tamaño 4x4. Al tomar estas dos salidas de 4x4, se termina así con un volumen de 4x4x2.

En general, si se tiene una imagen de entrada de $n \times n \times n_c$, donde el subíndice c indica el número de canales de color y se aplica una convolución con un filtro $f \times f \times n_c$ donde de nuevo debe ser el mismo n_c, entonces, lo que se obtiene es una salida de tamaño:

$$Imagen\ Salida = (n\ x\ n\ x\ n_C) * (f\ x\ f\ x\ n_C) = (n - f + 1)x(n - f + 1)$$

Que para el caso ilustrado en la figura 10 es:

$$Imagen\ Salida = (6\ x\ 6\ x\ 3) * (3\ x\ 3\ x\ 3) = (6 - 3 + 1)x(6 - 3 + 1) = 4x4$$

Esta expresión es con un stride o salto de 1 y sin relleno (padding p=0)

2.3.3 La arquitectura de una red neuronal convolucional

Una red neuronal que tiene una o varias capas convolucionales se llama Red Neuronal Convolucional (Convolutional Neural Network CNN). Consideremos un ejemplo (figura 11) de una red neuronal convolucional profunda para la clasificación de imágenes donde el tamaño de la imagen de entrada es 28 x 28 x 1 (escala de grises). En la primera capa, se aplica la operación de convolución con 32 filtros de 5 x 5 para que la primera salida sea de 24 x 24 x 32. Luego se aplica la agrupación con filtro 2 x 2 para reducir el tamaño a 12 x 12 x 32. En la segunda capa, se aplica la operación de convolución con 64 filtros de tamaño 5 x 5. Las dimensiones de salida se convertirán en 8 x 8 x 64 en las que se aplica la capa de agrupación con filtro 2 x 2 y el tamaño se reducirá a 4 x 4 x 64. Finalmente, se pasa a través de dos capas completamente conectadas (FC de Fully Connected Layers) para convertir la matriz de imagen de entrada a la red neuronal en una matriz de clasificación (Moolayil, 2019).

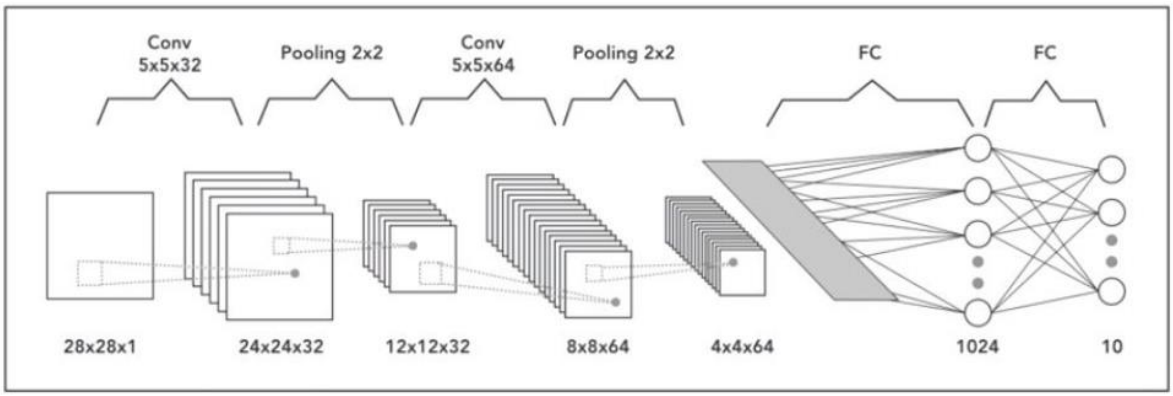


Figura 11. Arquitectura de una Red Neuronal Artificial (CNN)

Fuente: https://engmrk.com/convolutional-neural-network-3/

La estructura de datos fundamental en las redes neuronales es la capa (layer) que es un módulo de procesamiento de datos que toma como entrada uno o más tensores y genera uno o más tensores. Se puede pensar en las capas de una Red Neuronal como los ladrillos LEGO del D.L., una metáfora que es explícita por arquitecturas como Keras (Chollet, 2018). La construcción de modelos de D.L. en Keras se realiza recortando capas compatibles para formar pipelines útiles de transformación de datos. La noción de compatibilidad de capas aquí se refiere específicamente al hecho de que cada capa solo aceptará tensores de entrada de una determinada forma y devolverá tensores de salida de una determinada forma.

2.3.3.1 Tipos de capas en una CNN

Hay muchos tipos de capas para construir una Red Neuronal Convolucional (CNN), pero las más comunes son (Rosebroock, 2017):

- Capa de Convolución (CONV)
- *Capa de* Activación, entre estas se encuentran ACT o RELU, donde se utiliza la correspondiente Funcion de Activación.
- *Agrupación o* Pooling (POOL).
-*Completamente conectada o* Fully-connected (FC)
- Batch normalization (BN)
- Dropout (DO)

Considere el siguiente ejemplo con la arquitectura Keras (Chollet, 2018):

Código 1. Usando keras.

```
1.  from keras import layers
2.  layer = layers.Dense(32, input_shape=(784,))
```

Aquí se está creando una capa que solo aceptará como tensores 2D de entrada donde la primera dimensión es 784 (el eje 0, la dimensión del lote, no está especificado y, por lo tanto, cualquier valor sería aceptado). Esta capa devolverá un tensor donde la primera dimensión se ha transformado en 32.

Por lo tanto, esta capa solo se puede conectar a una capa aguas abajo que espera vectores de 32 dimensiones como su entrada. Al usar Keras, este asume la compatibilidad, porque las capas que se agregan a los modelos se crean dinámicamente para que coincidan con la forma de la capa entrante. Por ejemplo (Chollet, 2018):

Código 2. Uso de keras para definir la arquitectura de una red.

```
1.  from keras import models
2.  from keras import layers
3.  model = models.Sequential()
4.  model.add(layers.Dense(32, input_shape=(784,)))
5.  model.add(layers.Dense(32))
```

La segunda capa no recibió un argumento de forma de entrada; en cambio, infirió automáticamente su forma de entrada como la forma de salida de la capa anterior.

Ahora, como segundo ejemplo de una arquitectura de red neuronal convolucional, se muestra en la figura 12 la primera capa de la arquitectura AlexNet que ganó el desafío de clasificación ImageNet 2012 y es en gran medida responsable del auge actual del D.L. aplicado a la clasificación de imágenes. Dentro de su artículo (Krizhevsky, 2012) documentó su arquitectura CNN de acuerdo con la figura 13.

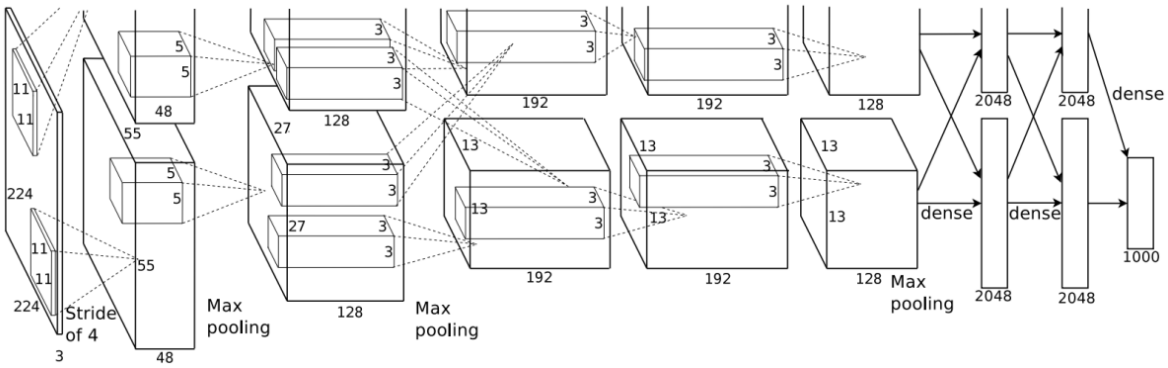


Figura 12. El diagrama original de la arquitectura AlexNet. Fuente: (Krizhevsky, 2012). Observe cómo se documenta que la imagen de entrada es 224 × 224 × 3.

Más adelante se tratarán otras arquitecturas de CNN.

En la siguiente sección se precisan cómo son estas capas de una CNN.

CAPAS DE AGRUPACIÓN (POOLING LAYERS): Aunque los parámetros totales de la red disminuyen después de la convolución, aún se requiere condensar aún más el tamaño espacial de la representación para reducir el número de parámetros y el cálculo en la red. La capa de agrupación hace este trabajo y acelera el cálculo, además de hacer que algunas de las características sean más destacadas. En la capa de agrupación, se tienen dos hiperparámetros de tamaño de filtro y stride (paso) que se establecen solo una vez. Por ejemplo en los Pooling Layers de 2x2 se reduce el tamaño a la mitad.

Los siguientes son dos tipos comunes de capas de agrupación: Max pooling y Average Pooling.

Max Pooling (Agrupación máxima): Consideremos una matriz de imagen de 4 x 4 que queremos reducir a 2 x 2. Para esto se usa un bloque de 2 x 2 con el tamaño de stride de 1. Se toma el valor máximo de cada bloque y se le captura en la nueva matriz, como se observa en la figura 13.

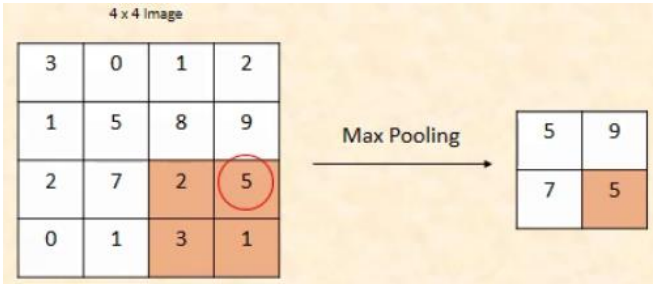


Figura 13. Ejemplo de aplicación de Max Pooling. Fuente: el autor.

Average Pooling: En la Agrupación Promedio (Average Pooling), se toma el promedio de cada uno de los bloques en lugar del valor máximo para cada uno de los bloques, que resulta para cada cuatro cuadrados en el caso de un Aerage Pooling de 2x2, como se ilustra en la figura 14

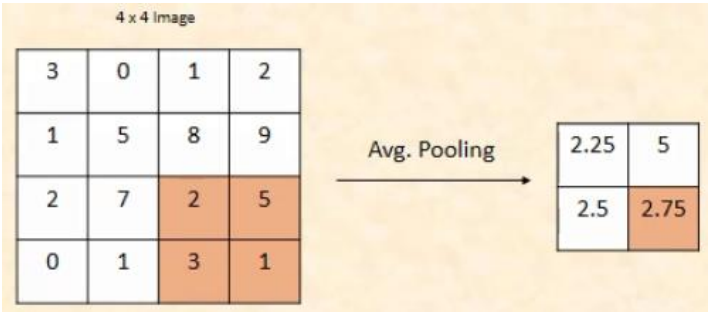


Figura 14. Ejemplo de Average Pooling. Fuente: el autor.

Capa Densa (Dense layer): Una capa densa es una capa normal de una Red Neuronal Profunda (DNN) que conecta cada neurona en la capa definida como densa con cada neurona en la capa anterior. Por ejemplo, si la Capa 1 tiene 5 neuronas y la Capa 2 (capa densa) tiene 3 neuronas, el número total de conexiones entre la Capa 1 y la Capa 2 sería 15 (5 × 3). Al adaptarse a todas las conexiones posibles entre las capas, se denomina capa "densa".

Capa de abandono (Dropout layer): La capa de abandono en D.L. ayuda a reducir el sobreajuste al introducir capacidades de regularización y generalización en el modelo. La capa de abandono elimina algunas neuronas o las establece en 0 y reduce el cálculo en el proceso de entrenamiento.

2.3.3.2 Otros elementos en una CNN

Otros elementos y funciones fundamentales para el entrenamiento de una CNN incluyen la función de activación, la función de pérdida (Loss Function) y las métricas para establecer el desempeño de la red entrenada.

Función de activación: Una función de activación es la función que toma la entrada combinada, aplica una función sobre esta y pasa el valor de salida, tratando de imitar la función de activación / desactivación.

La función de activación, por lo tanto, determina el estado de una neurona calculando la función de activación en la entrada combinada.

Hay una variedad de opciones disponibles para usar como función de activación. Las más comunes son la función sigmoidea y la ReLU (unidad lineal rectificada)

Función sigmoide (Sigmoid Activation Function): Históricamente, la función sigmoide es la función de activación más antigua y popular. Se define como:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Donde *e* es el número de Euler. Una neurona que utiliza la sigmoide como función de activación se le llama *neurona sigmoide*. Al establecer la variable z como la suma ponderada de entrada y después pasarla a través de la función sigmoide, se tiene:

$$z = \sum_i w_i x_i + b$$

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Que al graficar se obtiene una forma como la de la figura 15.

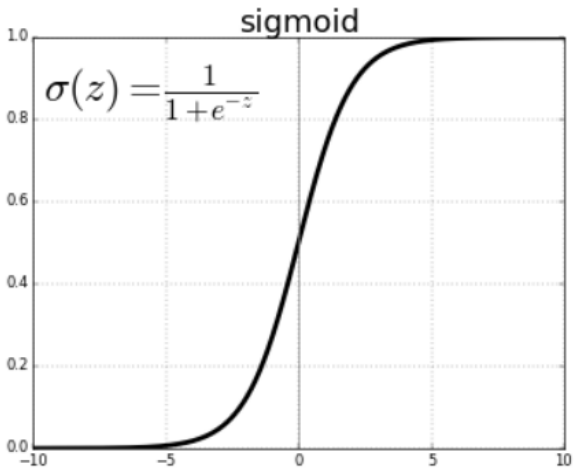


Figura 15. Función de activación sigmoide. Fuente:
https://www.experfy.com/blog/activation-functions-within-neural-networks

Se puede observar que a función de activación sigmoide actúa como una especie de función “aplanadora”, comprimiendo la salida a un rango entre 0 y 1.

Función RELU: La mayoría de las redes neuronales actuales usan otro tipo de función de activación llamada *rectified linear unit* o ReLU. Se define simplemente como

$$R(z)=max(0,z)$$

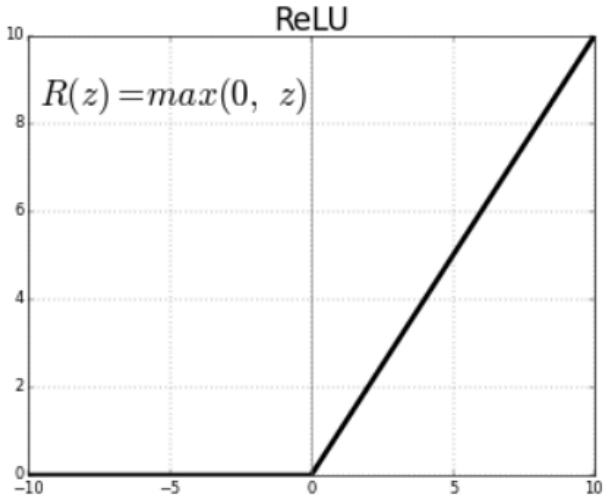


Figura 16. Función RELU. Fuente:

https://www.experfy.com/blog/activation-functions-within-neural-networks

Se observa que las funciones de activación ReLUs permiten el paso de todos los valores positivos sin cambiarlos y a los negativos les asigna el valor cero.

Función de pérdida (Loss Function): La función de pérdida es la métrica que ayuda a una red a comprender si está aprendiendo en la dirección correcta. Como analogía se puede considerar en un individuo los puntajes obtenidos en exámenes como indicador del proceso de aprendizaje por ejemplo en un idioma. De esta forma, suponga que una persona obtuvo 56, 60, 78, 90 y 96 de 100 en cinco exámenes de idioma consecutivos. Se vería claramente que los puntajes de las pruebas de mejora son una indicación de qué tan bien se está desempeñando. Si los puntajes de las pruebas hubieran disminuido, el veredicto sería que el desempeño de esta persona está disminuyendo y que necesitaría cambiar sus métodos o materiales de estudio para mejorar.

De forma similar, una red en proceso de aprendizaje utiliza la función de pérdida para chequear que está mejorando en cada iteración.

Algunas funciones de pérdida populares son:

- Error cuadrático medio (Mean Squared Error), es el promedio de la diferencia de los cuadrados entre los valores real y el predicho.
- Error absoluto medio (Mean Absolute Error), es a media absoluta entre el alor real y el predicho.

Para resultados categóricos, su predicción sería para una clase, como si un estudiante aprobará (1) o reprobará (0). Algunos casos de uso pueden tener múltiples clases como resultado, como clasificar tipos de células (Tipo Glóbulo rojo glóbulo blanco o Plaqueta), clasificar imágenes como los leucocitos entre sus tipos: neutrófilo, linfocito, monocito o eosinófilo, etc.

En tales casos, las pérdidas definidas en el precedente no pueden usarse debido a razones obvias. Se necesitaría cuantificar el resultado de la clase como probabilidad y definir pérdidas basadas en las estimaciones de probabilidad como predicciones.

Algunas opciones populares para pérdidas por resultados categóricos en Keras y otras arquitecturas, son las siguientes:

- Entropía cruzada binaria: define la pérdida cuando los resultados categóricos son una variable binaria, es decir, con dos resultados posibles: (Pasa / Falla) o (Sí / No).
- Entropía cruzada categórica (Categorical cross-entropy): define la pérdida cuando los resultados categóricos son no binarios, es decir, para más de dos clases o tipos de resultados posibles: (Sí / No / Quizás) o (Tipo 1 / Tipo 2 / ... Tipo n).

Optimizadores: El optimizador (Moolayil, 2019) es la parte más importante del entrenamiento de un modelo. Al tomar la estructura de un modelo definido para clasificar si una imagen es de tipo A o B, la estructura se crea definiendo la secuencia de capas con el número de neuronas, las funciones de activación y la forma de entrada y salida se inicializa con pesos aleatorios. La red actualiza los pesos que determinaron la influencia de una neurona en la siguiente neurona o la salida final durante el proceso de aprendizaje. Esto es, una red con pesos aleatorios y una estructura definida corresponde al punto de partida para un modelo.

La red toma una muestra de entrenamiento y usa sus valores como entradas para las neuronas en la primera capa, que luego produce una salida con la función de activación definida. La salida ahora se convierte en una entrada para la siguiente capa, y así sucesivamente. El resultado de la capa final sería la predicción para la muestra de entrenamiento. Aquí es donde entra en juego la función de pérdida. La función de pérdida ayuda a la red a comprender qué tan bien o mal se ha desempeñado el conjunto actual de pesos en la muestra de entrenamiento.

El siguiente paso para el modelo es reducir la pérdida. La función optimizadora le ayuda a determinar los pasos o actualizaciones que se deben realizar en los pesos para reducir la pérdida. La función optimizadora es un algoritmo matemático que utiliza derivadas, derivadas parciales y la regla de la cadena en el cálculo para comprender cuánto cambio verá la red en la función de pérdida al hacer un pequeño cambio en el peso de las neuronas.

El cambio en la función de pérdida, que sería un aumento o disminución, ayuda a determinar la dirección del cambio requerido en el peso de la conexión. El

cálculo de una muestra de entrenamiento desde la capa de entrada a la capa de salida se denomina pase. Por lo general, el entrenamiento se realizaría en lotes (batches) debido a restricciones de memoria en el sistema.

Un lote es una colección de muestras de entrenamiento de toda la entrada. La red actualiza sus pesos después de procesar todas las muestras en un lote. Esto se llama una iteración (es decir, un pase exitoso de todas las muestras en un lote seguido de una actualización de peso en la red). La computación de todas las muestras de entrenamiento proporcionadas en los datos de entrada con actualizaciones de peso lote por lote se denomina época (epoch). En cada iteración, la red aprovecha la función optimizadora para hacer un pequeño cambio en sus parámetros de peso (que se inicializaron aleatoriamente al principio) para mejorar la predicción final al reducir la función de pérdida. Paso a paso, con varias iteraciones y luego varias épocas, la red actualiza sus pesos y aprende a hacer una predicción correcta para las muestras de entrenamiento dadas.

Algunos algoritmos de optimización son el Gradiente de Descenso Estocástico (Stochastic Gradient Descent SGD).

En general un *gradiente* se define como la generalización de la derivada de una función en una dimensión a una función *f* en varias dimensiones. Se representa como un vector de *n* derivadas parciales de la función *f*. Es útil en la optimización porque el gradiente apunta en la dirección de la mayor tasa de aumento de la función para la cual la magnitud es la pendiente de la gráfica en esa dirección.

El Gradiente de descenso calcula la pérdida general a través de todos los ejemplos de entrenamiento antes de calcular el gradiente y actualizar el vector de parámetros. El SGD realiza una iteración con cada muestra de entrenamiento (es decir, después del paso de cada muestra de entrenamiento, calcula la pérdida y actualiza el peso) (Moolayil, 2019). Dado que los pesos se actualizan con demasiada frecuencia, la curva de pérdida general sería muy ruidosa. Sin embargo, la optimización es relativamente rápida en comparación con otras.

La fórmula para la actualización de los pesos se puede expresar como:

Pesos = Pesos - tasa de aprendizaje * Pérdida

En que la tasa de aprendizaje es un parámetro que se debe definir en la arquitectura de la red, algunos autores prefieren valores pequeños como 0.01.

Optimizador Adam: Adam, que significa Adaptive Moment Estimation, es uno de los optimizadores más populares y ampliamente utilizado en D.L..

Esta técnica de optimización calcula una tasa de aprendizaje adaptativo para el parámetro. Define el impulso y la varianza del gradiente de la pérdida y aprovecha un efecto combinado para actualizar los parámetros de peso. El

impulso y la varianza juntos ayudan a suavizar la curva de aprendizaje y mejorar efectivamente el proceso de aprendizaje.

Métricas: Las métricas se pueden entender como la función utilizada para juzgar el rendimiento del modelo en un conjunto de datos invisible diferente, también llamado conjunto de datos de validación. La única diferencia entre las métricas y la función de pérdida es que los resultados de las métricas no se usan para entrenar el modelo con respecto a la optimización. Solo se usan para validar los resultados de la prueba mientas se realiza el reporte de cómo va el entrenamiento de un modelo.

Con todos los elementos ya tratados de una red neuronal, se va a mostrar con el siguiente ejemplo cómo se crea un DNN en Keras con dos capas ocultas, con 32 y 16 neuronas, respectivamente, con una función de activación ReLU. El resultado final es para un resultado numérico categórico con cuatro posibles resultados como es el caso de clasificador que se va a desarrollar en este TFM para los leucocitos. Se utiliza una activación sigmoidea. Se compila el modelo con el optimizador Adam y se define la entropía cruzada como la función de pérdida y la exactitud (accuracy) como la métrica para la validación.

Código 3. Definiendo el modelo en keras con el optimizador Adam.

```
1.  from keras.models import Sequential
2.  from keras.layers import Dense, Activation
3.  model = Sequential()
4.  model.add(Dense(32, input_dim=10,activation = "relu"))
5.  model.add(Dense(16,activation = "relu"))
6.  model.add(Dense(4,activation = "sigmoid"))
7.  model.compile(optimizer='Adam',loss='categorical_crossentropy',
8.  metrics=['accuracy'])
```

2.4 Arquitecturas de CNN

En esta parte se mencionan algunas de las arquitecturas de red o modelos más modernos y varios de estos se utilizan en el entrenamiento de las redes que se ha propuesto en este TFM.

2.4.1 Arquitecturas o frameworks de Deep Learning de bajo nivel

Es necesario precisar que los términos en Inglés más aplicados aquí para describir este tipo de software son frameworks y libraries cuya mejor descripción es paquetes o módulos antes que propiamente arquitecturas. No obstante, estos dan lugar luego a otros programas más robustos y complejos que luego se mencionarán como frameworks de alto nivel que ya muchos autores y en la industria se les reconoce propiamente como arquitecturas. Así para esta primera parte se describirán como arquitecturas, paquetes, librerías o bibliotecas.

Dado el nivel de abstracción que proporciona una arquitectura de D.L., bien se las puede clasificar como de bajo o alto (Moolayil, 2019). Las siguientes son algunas de las arquitecturas o librerías más populares de bajo nivel para D.L.:

- Theano: una de las primeras librerías de D.L. en ganar popularidad desarrollada por el Instituto de Montreal para Algoritmos de Aprendizaje (MILA) en la Universidad de Montreal.

- Torch: es otra arquitectura de M.L. y D.L. basada en el lenguaje de programación Lua.

- PyTorch: es una librería de código abierto ML y DL para Python y fue desarrollada por el equipo de investigación de Inteligencia Artificia (AI) de Facebook.

- MxNet: fue desarrollado por investigadores de CMU, NYU, NUS, MIT y otros. La idea se simplificó para combinar programación declarativa e imperativa (mezclar) para maximizar la eficiencia y la productividad.

- TensorFlow: es una de las arquitecturas DL más populares y ampliamente utilizadas en la comunidad DL. Fue desarrollado y de código abierto por Google y también admite la implementación en CPU, GPU y dispositivos móviles. Fue lanzado en noviembre de 2015 y luego vio un gran aumento en su adopción dentro de la industria.

La lista de arquitecturas de D.L. es extensa y solo se mencionan otras muy populares como Caffe, Microsoft CNTK, Chainer, Paddle, etc. Discutir los pros y los contras de una arquitectura de DL sobre otra es un debate interesante e interminable y cada una en muchos casos tiene su propia comunidad de desarrolladores y programadores entusiastas.

2.4.2 Arquitecturas Deep Learning de alto nivel

Las arquitecturas mencionadas anteriormente se pueden definir como el primer nivel de abstracción para los modelos D.L. Aún se necesitaría escribir extensos códigos y scripts para preparar un modelo DL, aunque mucho menos que si se usa exclusivamente Python o C++. La ventaja de utilizar la abstracción de primer nivel es la flexibilidad que proporciona al diseñar un modelo.

Sin embargo, para simplificar el proceso de los modelos DL, se han desarrollado arquitecturas en el segundo nivel de abstracción; es decir, en lugar de usar directamente las arquitecturas mencionadas anteriormente, es posible usar un nuevo marco encima de un marco existente y así simplificar aún más el desarrollo del modelo D.L.

La arquitectura D.L. de alto nivel más popular que proporciona una abstracción de segundo nivel para el desarrollo del modelo D.L. es Keras. Otros marcos como Gluon, Lasagne, etc. también están disponibles, pero Keras ha sido el más ampliamente adoptado. Mientras que Gluon funciona encima de Mxnet, y lasagne encima de theano, Keras puede funcionar sobre tensorFlow, theano, Mxnet, Microsoft CntK y la lista sigue en aumento.

Keras es una API de red neuronal de alto nivel escrita en Python con la cual se puede desarrollar un modelo DL completamente funcional con pocas líneas de

código. Además, dado que admite varios otros marcos como back-end, agrega la flexibilidad para aprovechar una API de bajo nivel diferente para un caso de uso diferente si es necesario. La arquitectura más ampliamente adoptada de Keras es con TensorFlow como back-end,es decir, Keras como API DL de alto nivel y TensorFlow como back-end de API DL de bajo nivel. En las últimas versiones de tensorflow está keras altamente implementado e integrado con keras con comandos como tensorflow.keras o abreviadamente tf.keras de tal forma que la API de keras esté disponible.

2.4.3 Arquitecturas de CNN entrenadas con ImageNet

Entre estas arquitecturas se tiene a ResNet50 (desarrollado por Microsoft Research) o InceptionV3 (desarrollado por Google Research), listo para reconocer 1000 objetos comunes (lista de objetos ILSVRC).

Modelos existentes. VGG16 es una CNN propuesta por K. Simonyan y A. Zisserman de la Universidad de Oxford en el artículo “Redes convolucionales muy profundas para el reconocimiento de imágenes a gran escala”, lo cual no es el propósito de este TFM, en el cual se trabajará con un dataset de imágenes relativamente pequeño.

Desafortunadamente, hay dos inconvenientes principales con VGGNet, que se refieren especialmente a VGG-16 y VGG-19:

1. Es muy lento de entrenar.

2. Los pesos de la arquitectura de red en sí mismos son bastante grandes (en relación con el disco / ancho de banda).

Debido a su profundidad y número de nodos completamente conectados, VGG16 tiene más de 533 MB. Esto hace que la implementación de VGG sea una tarea agotadora. VGG16 se utiliza en muchos problemas de clasificación de imágenes de aprendizaje profundo; sin embargo, las arquitecturas de red más pequeñas a menudo son más deseables como son SqueezeNet, GoogLeNet, etc. Pero es un gran componente básico para el aprendizaje, ya que es fácil de implementar.

ResNet50 (Quora, s.f.) :t es el nombre abreviado para Red Residual y esta red presenta el Aprendizaje Residual.

Las CNN han llevado a una serie de avances para la clasificación de imágenes. Muchas otras tareas de reconocimiento visual también se han beneficiado enormemente de modelos muy profundos. Por lo tanto, a lo largo de los años hay una tendencia a seguir profundizando para resolver tareas más complejas mejorar la precisión de la clasificación de imágenes y reconocimiento de objetos. Pero a medida que se profundiza el entrenamiento de la red neuronal se torna de mayor dificultad y también la precisión comienza a saturarse y se degrada. El Aprendizaje Residual trata de resolver estos problemas.

En general, en una red neuronal convolucional profunda, se apilan varias capas y se entrenan para la tarea en cuestión. La red aprende varias características de niveles bajo, medio y alto al final de sus capas. En el Aprendizaje Residual, en lugar de tratar de aprender algunas características, se trata de aprender algo residual. Residual puede entenderse simplemente como la diferencia de la característica aprendida de la entrada de esa capa. ResNet realiza esta función mediante conexiones de acceso directo, es decir conectando directamente la entrada de la enésima capa a alguna capa n + x. Ha demostrado que entrenar esta forma de redes es más fácil que entrenar CNN profundas simples y también se resuelve el problema de la degradación de la precisión.

ResNet50 es una red residual de 50 capas. Hay otras variantes como ResNet101 y ResNet152.

2.5 Detección de Objetos

El D.L. ha impactado casi todas las facetas de la visión por computadora que se basa en el M.L. de una manera significativa, como son la Clasificación de imágenes, motores de búsqueda de imágenes, localización y mapeo (SLAM), segmentación de imágenes y por supuesto la Detección de Objetos (Rosebroock, 2017).
Uno de los algoritmos de detección de objetos basados en D.L. más populares es la familia de algoritmos R-CNN, introducidos originalmente por Girshick et al. en 2013 [43]. Uno de los algoritmos más recientes que ha surgido es YOLO (You Look Only Look Once) y con el que se va a trabajar en este TFM.

La detección de objetos tiene tres objetivos principales: dada una imagen de entrada que deseamos obtener:
1. Una lista de cuadros delimitadores (bounding boxes), o las coordenadas (x, y) para cada objeto en una imagen.
2. Una etiqueta de clase asociada con cada cuadro delimitador
3. La puntuación de probabilidad / confianza asociada con cada cuadro delimitador y etiqueta de clase.

Figura 17. Algunas formas de visión por computador: Clasificación de imágenes (izquierda), clasificación con localización (centro) y detección de objetos en una imagen (derecha). Fuente: el autor.

En la figura 17 se observan las diferencias entre la Clasificación de imágenes (en la figura de la izquierda) en que se tiene el ejemplo de un leucocito tipo neutrófilo, en que solo se considera un objeto. Luego previo a la detección se debe 'localizar' el objeto (imagen central) en que se debe trazar un bounding

box o cuadrado alrededor del objeto. En la parte de la derecha se observa la detección de objetos, en que puede haber varios objetos, en este caso dos leucocitos, el mismo neutrófilo de la primera imagen y también un linfocito.
Para la localización de un objeto, se requiere que el programa calcule algunos parámetros como se indica en la figura 18.

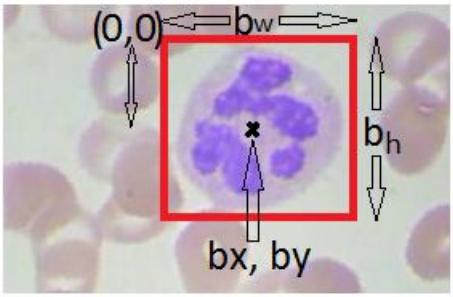


Figura 18. Parámetros para la localización de un objeto en una imagen. Fuente: el autor.

La red neuronal debe tener unas unidades de salida que generan un cuadro delimitador, en particular, puede tener la red neuronal de salida otras cuatro cantidades que son bx, by, bh, y bw, las cuales parametrizan el cuadro delimitador del objeto detectado.

Al utilizar la convención de notación que la parte superior izquierda de la imagen tiene las coordenadas (0,0) y en la parte inferior derecha está en (1,1). Entonces, al especificar el cuadro delimitador (el rectángulo rojo en la imagen) con el punto medio de coordenadas (b_x, b_y), la altura b_h y el ancho b_w.

El conjunto de entrenamiento para realizar la detección de objetos debe contener no solo la etiqueta de clase del objeto que la red neuronal va a tratar de predecir, y también debe contener estos cuatro números adicionales. Con el cuadro delimitador, puede usar aprendizaje supervisado para hacer su algoritmo que produce no solo una etiqueta de clase sino también los cuatro parámetros para indicarle dónde está el cuadro delimitador del objeto que detectó.

Entonces el algoritmo debe definir la etiqueta de destino, con base en aprendizaje supervisado, que para el caso desarrollado en este TFM son las tres clases (glóbulos rojos, glóbulos blancos y plaquetas) y la red neuronal ahora genera esos cuatro números, así como una etiqueta de clase, o las probabilidades de las etiquetas de clase. Así, adicional al proceso típico en una red neuronal para clasificación, en la localización de objetos se debe incluir en la salida las cuatro cantidades que determinan el Bounding box del objeto detectado en la imagen, como se muestra en la imagen 19.

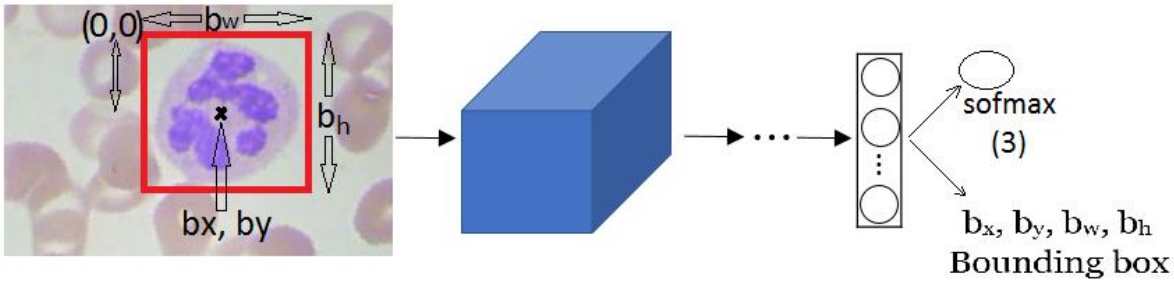


Figura 19. En una red de entrenamiento con localización,se incluye en la salida las cantidades que determinan el bounding box del objeto detectado. Fuente: el autor.

2.5.1 Definición de las etiquetas objetivo *y*

En el caso desarrollado en este TFM, el detector debe establecer a cuál de las siguientes clases pertenece el objeto (célula sanguínea), entre:

1. Glóbulo rojo o eritrocito.
2. Glóbulo blanco o leucocito.
3. Plaqueta.
4. Clase de fondo

Entonces, el algoritmo de detección de objetos define las cuatro cantidades b_x, b_y, b_w, b_h.
Correspondientes al bounding box donde haya detectado la célula, y así mismo la etiqueta de destino (la clase correspondiente a la célula detectada en una imagen) con cualquiera de los números 1, 2, 3 o si no hay tal célula entonces 4 (fondo de imagen) como se haya definido. También puede producir un vector de probabilidades para cada una de las clases a las que podría pertenecer la célula detectada.

Luego un vector de las etiquetas objetivo y se puede establecer como

$$y = \begin{pmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{pmatrix}$$

En este vector, p_c el primer componente del vector será la probabilidad de que haya un objeto, es decir, cualquiera de las clases 1, 2 o 3. En tal caso p_c será igual a 1, o si es la clase de fondo, entonces realmente no está tratando de detectar ninguno de los objetos, por lo que p_c sería 0.

A continuación, en caso de que haya un objeto, entonces se deben generar b_x, b_y, b_w, b_h, del cuadro delimitador o bounding box para el objeto detectado. Finalmente, si hay un objeto, es decir, si p_c es igual a 1, se debe indicar también en la salida los valores para c_1, c_2 y c_3 que especifica si es la

clase 1, clase 2 o clase 3. De esta manera si se detectó un leucocito, entonces p_c =1, c_1 =0, c_2 =1 y c_3 =0 y por supuesto también se especificarán los valores del cuadro delimitador b_x, b_y, b_w, b_h. Si no se detecta ningún objeto, entonces p_c =0 y todos los demás elementos del vector quedan indefinidos, que algunos autores en este último caso, escriben como signos de interrogación.

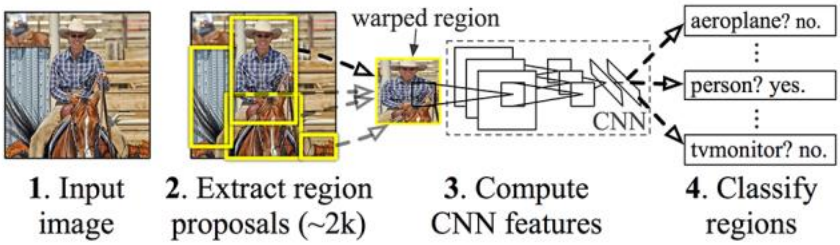


Figura 20. La arquitectura original de R-CNN consistía en (1) aceptar una imagen de entrada, (2) extraer alrededor de 2000 propuestas a través del algoritmo de búsqueda selectiva, (3) extraer características para cada Región de Interés (ROI) propuesta utilizando una CNN pre-entrenada y (4) clasificando las características para cada ROI usando un SVM lineal específico de la clase. Fuente: (Girshick, 2013)

2.5.2 Los algoritmos R-CNN

Una visión general del algoritmo R-CNN original se observa en la figura 20 que incluye un proceso de cuatro pasos:

• Paso # 1: ingresar una imagen

• Paso # 2: Extraer propuestas de regiones, es decir, partes de la imagen que potencialmente contienen objetos utilizando un algoritmo como la Búsqueda selectiva.

• Paso 3: utilizar el aprendizaje de transferencia, específicamente la extracción de características, para calcular características para cada propuesta que es un ROI (Region Of Interest) usando la CNN pre-entrenada.

• Paso 4: clasificar cada región propuesta utilizando las características extraídas con una máquina de vectores de soporte (SVM).

El problema con el enfoque original de R-CNN es que todavía es muy lento y realmente no se está aprendiendo a localizar la imagen a través de una red neuronal profunda.

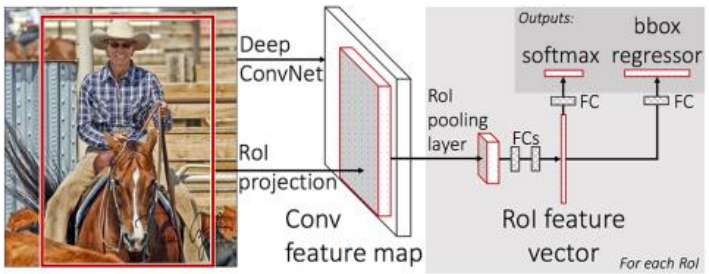


Figura 21. En la arquitectura Fast R-CNN, todavía se utiliza la Búsqueda selectiva para obtener las regiones propuestas, pero ahora se aplica el ROI Pooling extrayendo una ventana de tamaño fijo del mapa de características y utilizando estas características para obtener la etiqueta de clase final y el límite caja. Fuente: (Girshick R. B., 2015)

En Fast R-CNN que apareció un año después, se tiene que el beneficio principal es que la red es ahora entrenable desde el principio:

1. Se ingresa una imagen y cuadros de límite de verdad asociados (ground-truth bounding boxes).
2. Se extrae el mapa de características.
3. Se aplica la agrupación de ROI y se obtiene el vector de características de ROI.
4. Y finalmente se usan dos conjuntos de capas completamente conectadas para obtener (1) las predicciones de la etiqueta de clase y (2) las ubicaciones de los cuadros delimitadores para cada propuesta.

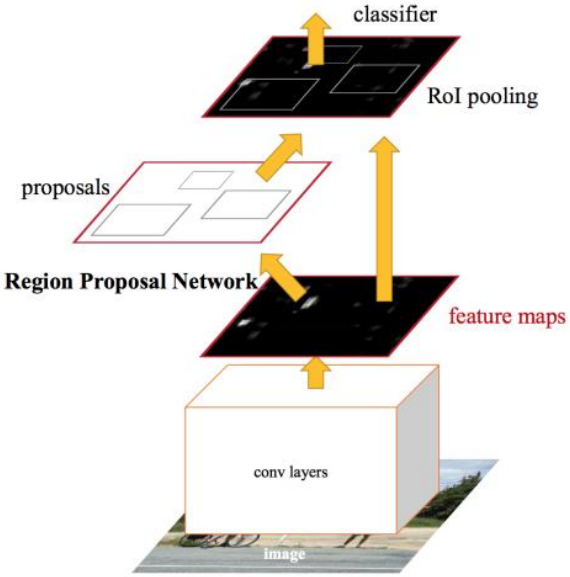


Figura 22. Faster R-CNN. En este algoritmo se presenta una Red de Propuesta de Región (RPN) que genera la propuesta de región directamente en la arquitectura. Fuente: (Girshick R. B., 2015)

En el Faster R-CNN (Girshick R. B., 2015). crearon un componente adicional para la arquitectura R-CNN que es una Red de Propuesta de Región (RPN – Region Proposed Region)- El objetivo de la RPN es eliminar la Búsqueda selectiva antes de la inferencia y, en su lugar, generar la región propuesta directamente en la arquitectura R-CNN.

Se han revisado brevemente la arquitectura Faster R-CNN, junto con sus variantes anteriores ya que estos han sido uno de los principales avances en Visión por computador. La arquitectura R-CNN ha presentado algunas iteraciones y mejoras, y con la última arquitectura Faster R-CNN se puede entrenar detectores de objetos de D.L. de inicio a fin.

La arquitectura en sí incluye cuatro componentes principales:
- el primer componente es la red base (es decir, ResNet, VGGNet, etc.) que se utiliza como un extractor de funciones.
- luego se tiene la Red de Propuestas de Región (RPN), que acepta un conjunto de anclas y genera propuestas sobre dónde es posible que estén los objetos en una imagen. Es importante precisar que el RPN no sabe cuál es el objeto en la imagen, solo que existe un objeto potencial en una ubicación determinada.
- La agrupación de regiones de interés se utiliza para extraer mapas de características de cada región propuesta.

- El cuarto elemento es una red neuronal convolucional basada en la región para (1) obtener las predicciones finales de la etiqueta de clase para la propuesta y (2) refinar aún más las ubicaciones de la propuesta para una mayor precisión.

Dado el gran número de partes móviles en la arquitectura R-CNN, muchos autores no recomiendan implementar toda la arquitectura a mano. En cambio, se aconseja usar implementaciones existentes como la API de detección de objetos TensorFlow.

2.5.3 El algoritmo YOLO (You only look once)

El trabajo de Girchick et al. en sus tres publicaciones R-CNN es realmente notable y ha permitido que la detección de objetos basada en D.L. sea una realidad. Sin embargo, hay algunos problemas que tanto los investigadores como los profesionales encontraron con las R-CNN (Rosebroock, 2017).

La primera consiste en que el entrenamiento requirió múltiples fases. La RPN necesitaba entrenamiento para generar los cuadros delimitadores sugeridos antes de que se pudiera entrenar al clasificador real para el reconocimiento de objetos en las imágenes. El problema se mitigó luego al entrena toda la arquitectura R-CNN desde el inicio, pero antes de este descubrimiento, requirió un proceso de pre-entrenamiento.

El segundo problema es que el entrenamiento tardaba demasiado. Un Faster R-CNN tiene múltiples componentes como:

1. Una red de propuestas de la región (RPN).
2. Un módulo de agrupación de ROI (Región de Interés).
3. El clasificador final

Si bien los tres encajan en una arquitectura, todavía son partes móviles que ralentizan todo el procedimiento de entrenamiento.
El último problema es que el tiempo de inferencia fue demasiado lento: todavía no se podía obtener la detección de objetos en tiempo real con aprendizaje profundo.

YOLO es uno de los algoritmos de detección de objetos más rápidos basados en la red neuronal convolucional, utiliza una estrategia de detector de una etapa donde todo se realiza en una sola pasada. Este algoritmo se basa en la regresión y el algoritmo aprende simultáneamente las coordenadas del cuadro delimitador y las probabilidades correspondientes de la etiqueta de clase (Soto & Serrano, 2017).

YOLO divide la imagen en una cuadrícula de 13 por 13 celdas (imagen 22 a la izquierda). Cada celda es capaz de predecir 5 cuadros delimitadores y cada celda también predice una clase.

Por lo tanto, hay 13 * 13 * 5 = 845 cuadros delimitadores en total (imagen superior central).

Cada uno de los cuadros delimitadores tiene un puntaje de confianza que indica la certeza del cuadro delimitador. El puntaje de confianza indica que la forma del cuadro es buena o mala (imágenes centrales superior e inferior).
El puntaje de confianza y la predicción de clase se fusionan para derivar el puntaje final que indica la probabilidad de que un cuadro delimitador contenga un tipo específico de objeto medido contra los pesos etiquetados y entrenados. Por ejemplo, el cuadro azul de la izquierda es del 75%, lo que confirma que el objeto es "perro":

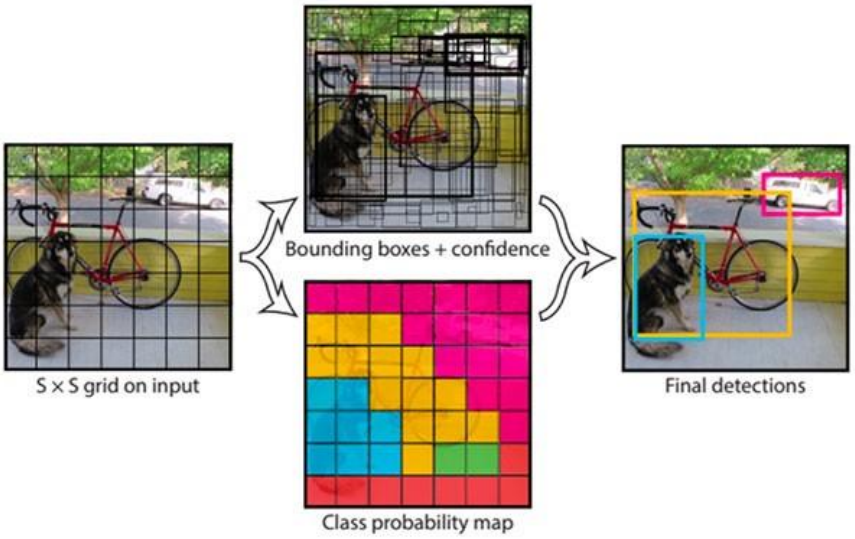


Figura 23. Detección de objetos con YOLO
Fuente: (Shankar, 2019)

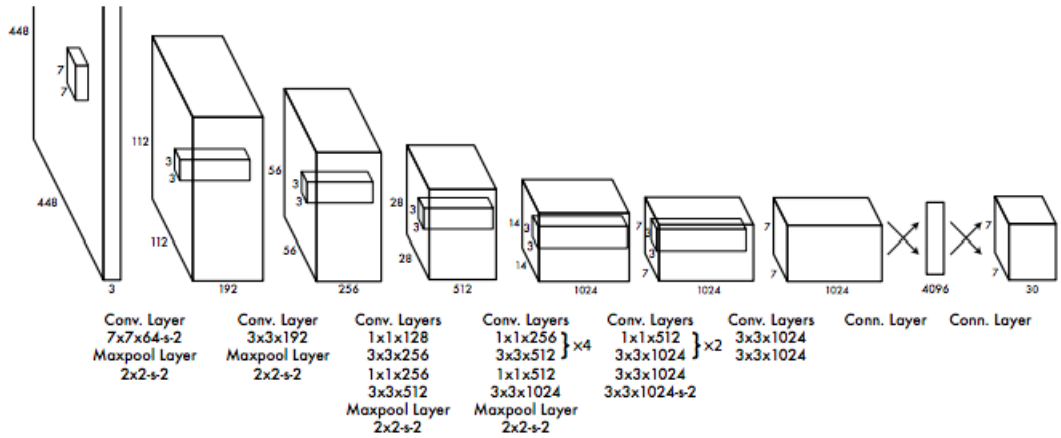


Figura 24 Arquitectura de Yolo. Fuente: (Soto & Serrano, 2017)

La arquitectura de la red YOLO contiene 24 capas convolucionales seguidos por dos capas totalmente conectadas (FC). Al alternar capas convolucionales de 1x1 se reduce el espacio de las características desde las capas anteriores. La red de capas convolucionales fue pre entrenada sobre el desafío de clasificación de ImageNet a la mitad de la resolución (224 x 224 de imágenes de entrada) y luego se duplicó la resolución para la detección.

2.5.4 La librería ImageAI

ImageAI es un contenedor fácil de usar para las arquitecturas de reconocimiento de imágenes de D.L. SqueezeNet, ResNet, InceptionV3 y DenseNet. ImageAI proporciona métodos muy convenientes y potentes para realizar la detección de objetos en imágenes y extraer cada objeto de la imagen. La clase de detección de objetos admite RetinaNet, YOLOv3 y TinyYOLOv3.
Se usó para el Detector la librería ImageAI con YOLO, de acuerdo con el código para su implementación (Olafenwa).

3. Materiales y métodos

Se van a seguir para la metodología los pasos para construir una red de D.L. que incluyen (Rosebroock, 2017):

- Paso 1: Recolectar el dataset.
- Paso 2: Dividir el dataset.
- Paso No. 3. Entrenar el modelo.
- Paso No. 4: Evaluar el modelo

3.1 Paso No. 1: Recolectar el Dataset

El primer componente de la construcción de una red de Aprendizaje Profundo es reunir el conjunto de datos inicial. Fue necesario realizar una intensa búsqueda y selección de los conjuntos de datos de imágenes de células de la sangre normales que sirvieran para los fines de este TFM. Por el modelo desarrollado que es supervisado, se requieren las imágenes en sí mismas, así como las etiquetas asociadas con cada imagen. Estas etiquetas deben provenir de un conjunto finito de categorías definidas para glóbulo rojo, glóbulo blanco o plaqueta en un primer nivel de clasificación. Luego en un segundo nivel, correspondiente al desarrollo del detector, deberá discriminar para los glóbulos blancos entre neutrófilo, eosinófilo, monocito y linfocito.

Adicionalmente el número de imágenes de cada categoría debe ser aproximadamente el mismo, en caso contrario, se presenta el conocido Desbalance y trae efectos indeseables como que el clasificador pueda quedar sesgado y sobreajustar las categorías con mayor cantidad de imágenes.

La obtención de un dataset apropiado para este TFM ha sido una labor dispendiosa, por lo cual se mencionan las etapas para la consecución del mismo.

1. Dataset inicial considerado y propuesto por el tutor para el estudio previo a la estructuración del TFM: Datos clínicos del laboratorio CORE del Hospital Clínic de Barcelona, en el cual a partir de la rutina diaria se han almacenado por más de 7 años, muestras de frotis de sangre periférica y se ha creado una base de datos de descriptores cuantitativos a partir de las imágenes de células, involucrando múltiples enfermedades.

Este dataset se descartó ya que implicaría trabajos utilizando técnicas bio-estadísticas y de M.L. para el análisis y clasificación de las diferentes estirpes celulares asociados a diferentes patologías, y básicamente al estructurar este TFM los objetivos se dirigieron a la detección y clasificación de células normales y no representativas de alguna patología. Adicionalmente el dataset no es de dominio público.

2. Dataset de células normales de la sangre de Kaggle. El sitio www.kaggle.com es un repositorio de datasets de dominio público para realizar específicamente M.L. Este dataset contiene 12,500 imágenes aumentadas de células sanguíneas (JPEG) con etiquetas de tipo celular (CSV). Hay aproximadamente 3.000 imágenes para cada uno de los 4 tipos de células

diferentes agrupados en 4 carpetas diferentes (según el tipo de célula). Los tipos de células son eosinófilos, linfocitos, monocitos y neutrófilos. Este conjunto de datos está acompañado por un conjunto de datos adicional que contiene las 410 imágenes originales (previas al aumento), y también los cuadros delimitadores (bounding boxes) para cada célula en cada una de estas 410 imágenes (metadatos JPEG + XML). Más específicamente, la carpeta 'dataset-master' contiene 410 imágenes de células sanguíneas con etiquetas de subtipo y cuadros delimitadores (JPEG + XML), mientras que la carpeta 'dataset2-master' contiene 2,500 imágenes aumentadas y 4 etiquetas de subtipo adicionales (JPEG + CSV).

Hay aproximadamente 3.000 imágenes aumentadas para cada clase de las 4 clases en comparación con 88, 33, 21 y 207 imágenes de cada una en la carpeta 'dataset-master'. Esta forma estructural del dataset es el apropiado para obtener los dos productos: el **clasificador** con la carpeta 'dataset2-master', y el **detector** con la carpeta 'dataset-master'. Luego este dataset se utilizó para entrenar ambos modelos. Para el **Clasificador** se utilizó también el dataset de LISC como se detalla a continuación. La licencia es MIT (https://github.com/Shenggan/BCCD_Dataset).

Para el clasificador, se usa la carpeta 'dataset2-master', con la estructura dada por:

|---dataset2-master
|---|---images
|---|---|TEST
|---|---|---|EOSINOPHIL
|---|---|---|LYMPHOCYTE
|---|---|---| MONOCYTE
|---|---|---|NEUTROPHIL
|---|---|TRAINING
|---|---|---|EOSINOPHIL
|---|---|---|LYMPHOCYTE
|---|---|---| MONOCYTE
|---|---|---|NEUTROPHIL

Estas imágenes son de tamaño 320x240 pixeles y entre 7 a 9KB. Se muestra una de las figuras de la carpeta TRAINING/NEUTROPHIL

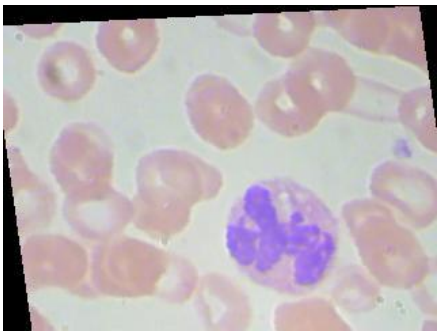


Figura 25. Imagen de la carpeta dataset2-master/ TRAINING/NEUTROPHIL de las imágenes de Kaggle.com

La carpeta 'dataset-master' se utiliza para el entrenamiento del Detector. La estructura es:

|---dataset-master
|---|---Testing
|---|---| Annotations
|---|---|---BloodImage000.xml
…
|---|---|Images
|---|---|---BloodImage000.JPEG

…
|---|---Training
|---|---| Annotations
|---|---|---BloodImage000.xml
…
|---|---|Images
|---|---|---BloodImage000.JPEG

…
|---|---Validation
|---|---| Annotations
|---|---|---BloodImage000.xml
…
|---|---|Images
|---|---|---BloodImage000.JPEG

…
Estas imágenes tienen un tamaño de 640x480 pixeles y de aproximadamente 23KB. Se muestra una imagen en la figura 25.

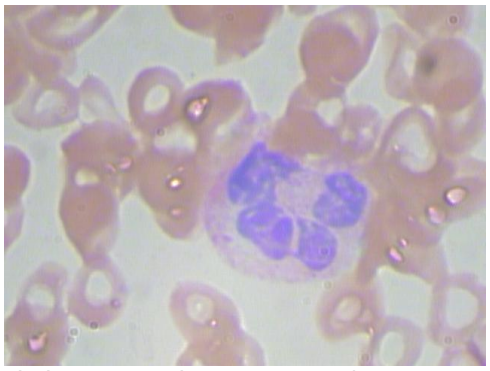


Figura 26. Imagen de la carpeta dataset-master/ JPEGImages de Kaggle.com

- Dataset Leukocyte Images for Segmentation and Classification -LISC.
La base de datos LISC incluye las imágenes hematológicas tomadas de sangre periférica de sujetos sanos. La base de datos se ha lanzado para permitir la evaluación comparativa de diferentes técnicas en núcleo y segmentación de citoplasma y también reconocimiento de diferentes glóbulos blancos en imágenes hematológicas.

Se tomaron muestras de sangre periférica de 8 sujetos normales y se obtuvieron 400 muestras de 100 portaobjetos de microscopio. Los portaobjetos del microscopio se untaron y se tiñeron con la técnica Gismo-Right y las imágenes se obtuvieron con un microscopio óptico (Microscope-Axioskope 40) de la sangre periférica teñida usando una lente acromática con un aumento de 100. Luego, estas imágenes se registraron mediante un digital cámara (Sony Modelo No. SSCDC50AP) y se guardaron en formato BMP. Las imágenes contienen 720 × 576 píxeles. Todas son imágenes en color y se obtuvieron del Centro de Investigación de Hematología-Oncología y BMT del hospital Imam Khomeini en Teherán, Irán. Las imágenes fueron clasificadas por un hematólogo en leucocitos normales: basófilos, eosinófilos, linfocitos, monocitos y neutrófilos. Además, las áreas relacionadas con el núcleo y el citoplasma fueron segmentadas manualmente por un experto.
Los datos incluidos en esta base de datos se pueden utilizar, de forma gratuita, con fines de investigación y educación.

Se reconoce en este TFM el uso de la base de datos LISC: (Rezatofighi & Soltanian-Zadeh,)

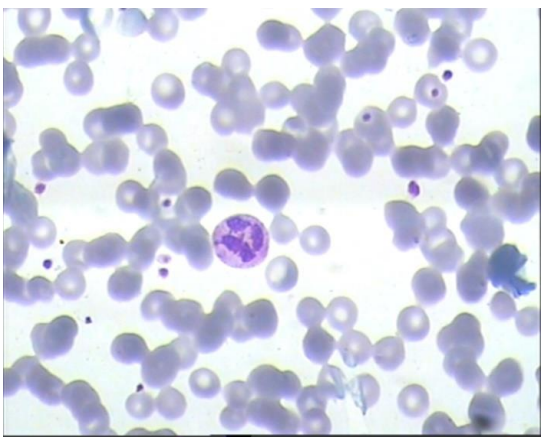


Figura 27. Imagen del dataset LISC

También se descargaron y revisaron otros datasets, pero en general se encontró que están enfocados a segmentación de imágenes, lo cual no es de interés en este trabajo.

Luego se encontró el dataset del estudio realizado por (Wang Q. , Sun, Wang, & Yang, 2019)**,** con el dataset disponible en el sitio : https://doi.org/10.6084/m9.figshare.8206625.v1.

No obstante, se encontró que si bien las imágenes son de altísima calidad, estas no están clasificadas ni etiquetadas para poder realizar el estudio que es de tipo Supervisado. Se muestra una imagen de este dataset.

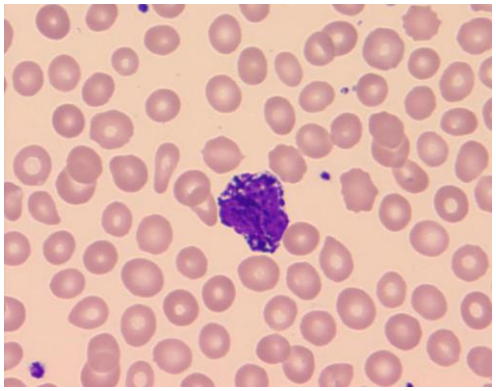


Figura 28. Imagen del dataset del estudio realizado por (Wang, Sun, & Yang, 2019)

El autor del presente TFM ha decidido con el apoyo del tutor, utilizar otros datasets como el de Kaggle para clasificación de las células normales de la sangre, fusionándolo con el dataset de LISC para lo cual se ha hecho el recorte de estas últimas al mismo tamaño de las imágenes de Kaggle.

Edición de las imágenes del Dataset LISC. Las imágenes originales tienen un tamaño de 720x576 y un peso de alrededor de 230 kB. Se recortan al tamaño de 320x240.

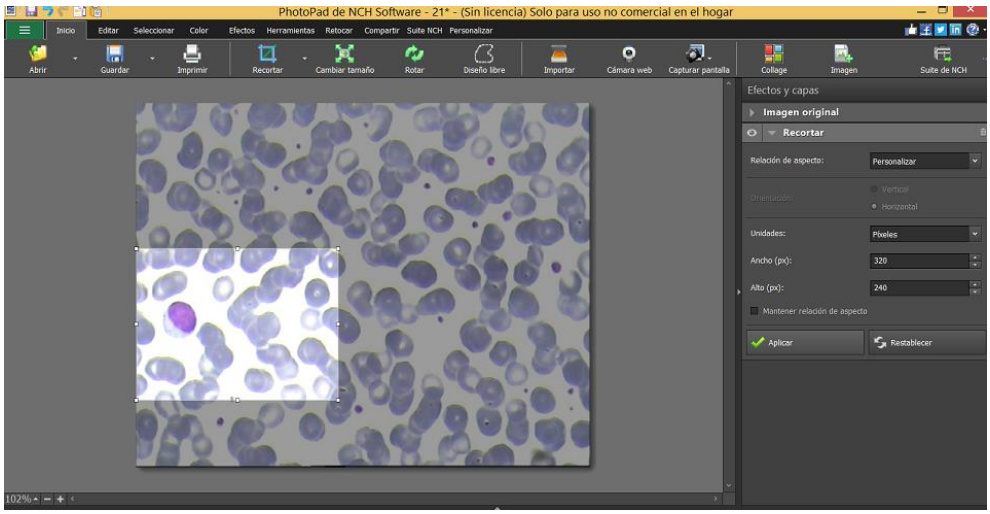


Figura 29. Edición de las imágenes del dataset LISC en el programa Photopad. Fuente: el autor.

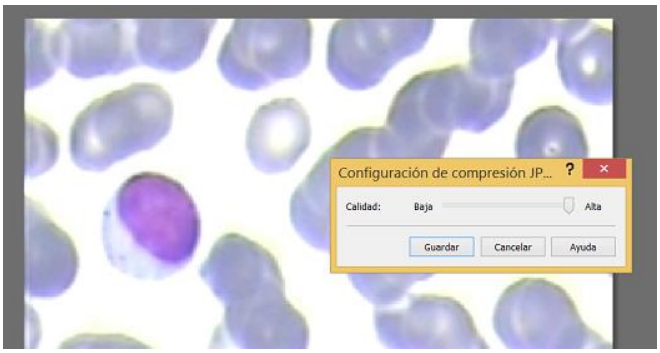


Figura 30. Imagen en resolución alta, que da un tamaño de archivo de alrededor de 30 kb. También se deja una imagen en resolución media que da un tamaño de alrededor de 6.5 kB. Fuente: el autor.

De esta forma se incrementan en alrededor de 100 imágenes el dataset total por cada una de las 4 categorías de leucocitos o glóbulos blancos que están en el dataset de Kaggle, a saber, eosinófilos, monocitos, linfocitos y neutrofilos, y no se tomaron en cuenta los otros tipos de células como los basófilos, así como otras categorías de células sanguíneas como las plaquetas y los glóbulos rojos. Con esto se busca aumentar la Generalización del modelo que se obtenga, ya que se tienen fuentes de imágenes diferentes.

3.2 Paso No. 2: Dividir el Dataset.

El dataset se debe dividir en al menos dos partes:

1. Un subconjunto de entrenamiento o *training set*, con alrededor del 80% del dataset.
2. Un subconjunto de pruebas o *testing set* con las demás imágenes y etiquetas.

Algunos modelos requieren un tercer subconjunto llamado de Validación.

El clasificador utiliza un conjunto de entrenamiento para "aprender" cómo se ve cada categoría realizando predicciones en los datos de entrada y luego se corrige cuando las predicciones son incorrectas. Después de que el clasificador ha sido entrenado, se evalúa el desempeño en un conjunto de pruebas, ambos subconjuntos deben ser independientes.

Ajuste de parámetros o Tuning: Las redes neuronales tienen una serie de ítems como velocidad de aprendizaje, decadencia, regularización, etc. que deben ajustarse y marcarse para obtener un rendimiento óptimo. Este tipo de parámetros son los hiperparámetros.

Se deben probar una buena cantidad de estos hiperparámetros e identificar el conjunto de parámetros que funciona mejor. También se puede crear una tercera división de datos llamada conjunto de validación. Este conjunto de datos proviene de los mismos datos de entrenamiento y se utiliza como "datos de prueba falsos" para que se puedan ajustar los hiperparámetros. Solo después de determinar los valores del hiperparámetro utilizando el conjunto de

validación, se pasa a recopilar resultados de precisión finales en los datos de prueba.

3.3. Paso No. 3. Entrenar los modelos Clasificado y Detector

El objetivo aquí es que la red aprenda a reconocer cada una de las categorías en los datos etiquetados. Cuando el modelo comete un error, aprende de este error y se mejora a sí mismo.

3.3.1 Entrenamiento del Clasificador

Para entrenar la red del Clasificador, se siguen las siguientes etapas:

- Configurar el servidor y carga del dataset. Con el lanzamiento por parte de google de Tensorflow 2.0 y su disponibilidad desde Google Colab, así como la posibilidad en este servidor de uso libre y gratuito de utilizar aceleradores como GPU y TPU, se ha trabajado por ahora con este servidor para estas primeras pruebas. Se han subido las imágenes del dataset al Drive de la cuenta de google del autor y que gestiona la UOC, ya que esta tiene una mayor capacidad que las cuentas independientes. Luego con Google Colab se ACTIVA DRIVE para poder abrir este dataset como se ilustra en la siguiente imagen.

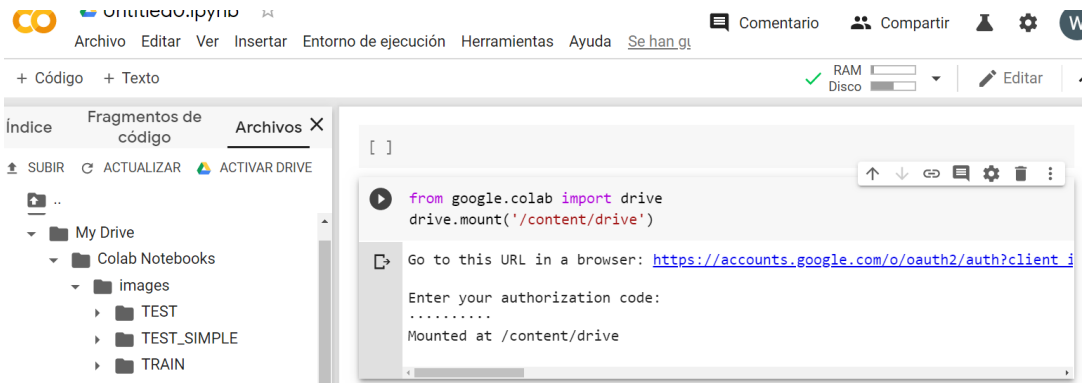


Figura 31. Activación de DRIVE en Google colab. Fuente: el autor.

Luego para cargar la versión más reciente de Tensorflow, se utiliza el siguiente código:

Código 4. Instalar la versión más reciente de tensorflow.

```
from __future__ import absolute_import, division, print_function, unicode_literals

# Install TensorFlow
try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass

import tensorflow as tf
```

Para revisar la versión de tensorflow, se digita:

```
tf.__version__
```

y se confirma que es

```
'2.1.0-rc1'
```

En la página oficial de Tensorflow (Google, s.f.) se pueden encontrar todas las versiones, las cuales presentan con frecuencia cambios sutiles en el código que se utiliza para entrenar una red en especial en el nombre de funciones, y esto por supuesto que va a generar errores:

En Google colab por defecto se carga la versión 1.15, aunque en este caso se advierte que próximamente la versión por defecto será la 2.0.0
Luego para acceder a la carpeta de las imágenes, se debe ser cuidadoso de seguir exactamente la ruta y definirla así en el modelo a entrenar como se observa en la figura 32:

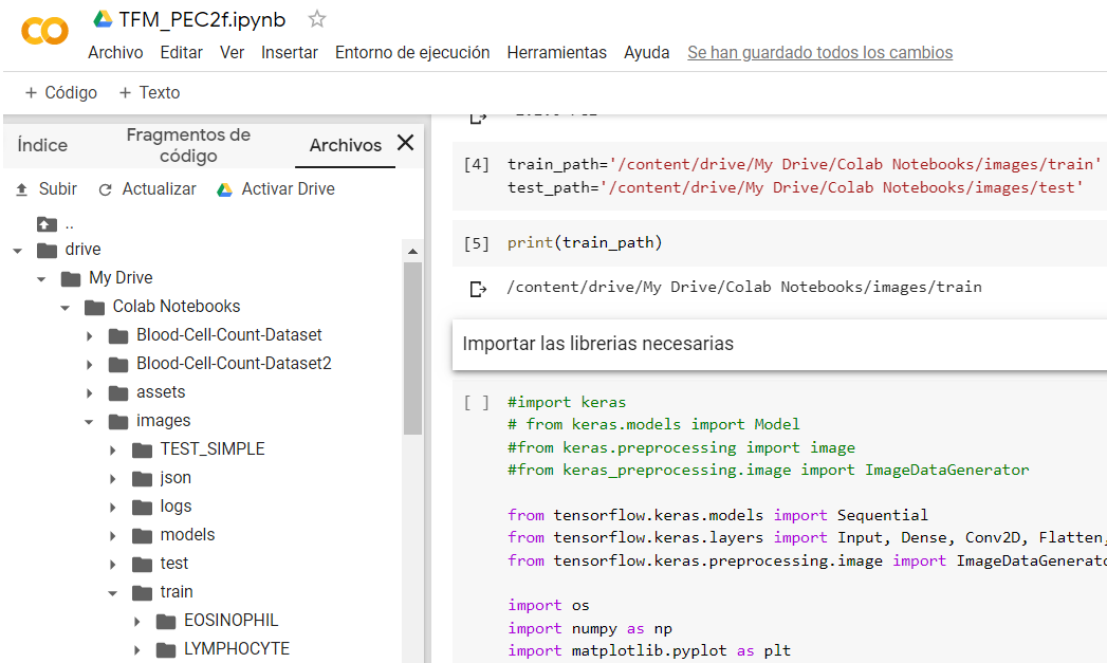


Figura 32. Estableciendo la ruta donde se encuentran las imágenes en Drive para entrenar la red en Google Colab. Fuente: el autor.

En el Notebook que se encuentra disponible en github, se observa cómo se lee el dataset y discrimina este entre las 4 categorías. También se muestran algunas imágenes.

Con el siguiente código se observa que reconoce la ruta indicada de las diferentes carpetas de las imágenes así como la distribución de cada categoría tanto en el conjunto de *train* como en el de *test.*

Código 5. Reconocer la ruta del dataset.

```
#Training:
eosinophil_dir = os.path.join('/content/drive/My Drive/Colab Notebooks/images/train/EOSINOPHIL')
lymphocyte_dir = os.path.join('/content/drive/My Drive/Colab Notebooks/images/train/LYMPHOCYTE')
monocyte_dir = os.path.join('/content/drive/My Drive/Colab Notebooks/images/train/MONOCYTE')
neutrophil_dir = os.path.join('/content/drive/My Drive/Colab Notebooks/images/train/NEUTROPHIL')

#Validation:
eosinophil_Vdir = os.path.join('/content/drive/My Drive/Colab Notebooks/images/test/EOSINOPHIL')
lymphocyte_Vdir = os.path.join('/content/drive/My Drive/Colab Notebooks/images/test/LYMPHOCYTE')
monocyte_Vdir = os.path.join('/content/drive/My Drive/Colab Notebooks/images/test/MONOCYTE')
neutrophil_Vdir = os.path.join('/content/drive/My Drive/Colab Notebooks/images/test/NEUTROPHIL')

print('total imagenes training eosinophil:', len(os.listdir(eosinophil_dir)))
print('total imagenes training lymphocyte:', len(os.listdir(lymphocyte_dir)))
print('total imagenes training monocyte:', len(os.listdir(monocyte_dir)))
print('total imagenes training neutrophil:', len(os.listdir(neutrophil_dir)))

print('total imagenes test eosinophil:', len(os.listdir(eosinophil_Vdir)))
print('total imagenes test lymphocyte:', len(os.listdir(lymphocyte_Vdir)))
print('total imagenes test monocyte:', len(os.listdir(monocyte_Vdir)))
print('total imagenes test neutrophil:', len(os.listdir(neutrophil_Vdir)))
```

El resultado que arroja es:

```
total imagenes training eosinophil: 2603
total imagenes training lymphocyte: 2589
total imagenes training monocyte: 2584
total imagenes training neutrophil: 2605
total imagenes test eosinophil: 629
total imagenes test lymphocyte: 626
total imagenes test monocyte: 626
total imagenes test neutrophil: 630
```

Se observa también que de la carpeta *train* hay 9957 imágenes mientras que en la carpeta *test* hay 2487.

Se utiliza el siguiente código para entender cómo desde Google Colab y con Python se manejan los archivos de imágenes, realizando una muestra de 6 imágenes de cada una de las categorías de las correspondientes a la carpeta *train*:

Código 6. Manejo de los archivos de imágenes en Python con Google colab.

```
eos_files = os.listdir(eosinophil_dir)
print('eos:',eos_files[:6])
lympho_files = os.listdir(lymphocyte_dir)
```

```
print('limpho: ',lympho_files[2373:2385])
mono_files = os.listdir(monocyte_dir)
print('mono: ',mono_files[2372:2383])
neutro_files = os.listdir(neutrophil_dir)
print('neutrop:', neutro_files[2390:2399])
```

con lo cual se obtiene (se edita con puntos suspensivos para abreviar):

```
eos: ['_53_1628.jpeg', '_53_1261.jpeg', …, '_53_2625.jpeg']
limpho:  ['_13_1349.jpeg', '_13_1364.jpeg', … '_13_2696.jpeg']
mono:  ['_11_4593.jpeg', '_11_4104.jpeg', …, '_11_5482.jpeg']
neutrop: ['_121_5471.jpeg', '_121_7410.jpeg',…, '_122_5784.jpeg']
```

También se realiza una vista de algunas imágenes directamente desde python con el notebook con el siguiente código:

```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
pic_index = 2
next_eos = [os.path.join(eosinophil_dir, fname)
                for fname in eos_files[pic_index-2:pic_index]]
next_lympho = [os.path.join(lymphocyte_dir, fname)
                for fname in lympho_files[pic_index-2:pic_index]]
next_mono = [os.path.join(monocyte_dir, fname)
                for fname in mono_files[pic_index-2:pic_index]]
next_neutro = [os.path.join(neutrophil_dir, fname)
                for fname in neutro_files[pic_index-2:pic_index]]
for i, img_path in enumerate(next_eos+next_lympho+next_mono+next_neutro):
  #print(img_path)
  img = mpimg.imread(img_path)
  plt.imshow(img)
  plt.axis('Off')
  plt.show()
```

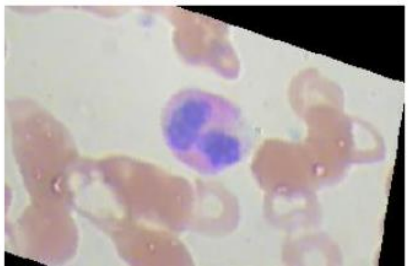


Figura 33. Visualización de algunas imágenes en el notebook. Fuente: el autor.

- Realizar el Machine learning. Se ejecutaron varias arquitecturas tipo Sequential, en las primeras el accuracy rondó por el 25% por lo cual no se realizó una verdadera clasificación de las imágenes, ya que al ser de 4 tipos de categorías la probabilidad de acertar una imagen en particular al azar es del 25%. Estas son los primeros experimentos, y en el Notebook se muestran las arquitecturas probadas. Se muestran los resultados de algunas de estas corridas con los siguientes códigos e imágenes.

Código 7. Preparando el dataset para el modelo 3.

```
#Modelo 3 En este se realiza previamente un procesamiento de imágenes
from tensorflow.keras.preprocessing import image

generator = image.ImageDataGenerator(
        rescale = 1./255,
```

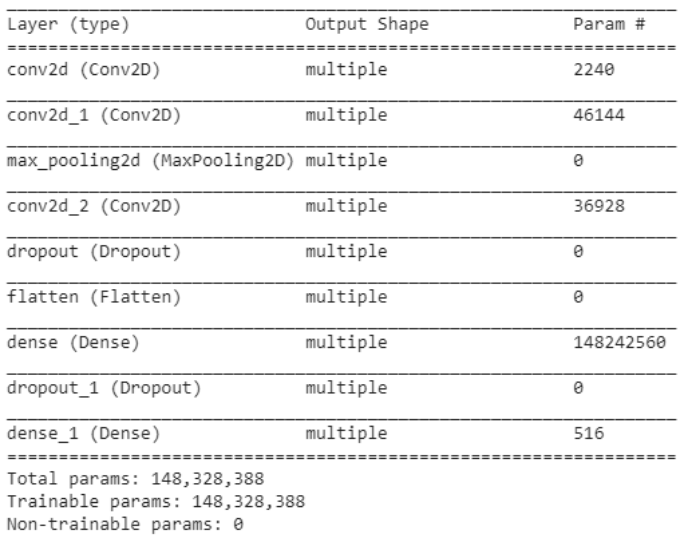
```
        featurewise_center=False,  # establecer la media a 0 sobre el dataset

        samplewise_center=False,  # establecer cada media muestral a 0
        featurewise_std_normalization=False,  # dividir las entradas por la de
sviacion estandar del dataset
        samplewise_std_normalization=False,  # dividir cada entrada por su des
viación estandar
        zca_whitening=False,  # apilicar el ZCA whitening
        rotation_range=10,  # rotar las imágenes aleatoriamente en el rango (g
rados deg, 0 to 180)
        width_shift_range=0.1,  # aleatoriamente cambiar las imagenes horizont
almente (fraccion del ancho total)
        height_shift_range=0.1,  # aleatoriamente cambiar las imágnes vertical
ente (fraccion de la altura total)
        horizontal_flip=True,  # aleatoriamente voltear las imágenes
        vertical_flip=False)
dataset = generator.flow_from_directory(
    shuffle = True,
    batch_size = 32, #No es necesario
    target_size = (240, 320),
    directory = '/content/drive/My Drive/Colab Notebooks/images/TRAIN'
)
```

Luego se establece la arquitectura del modelo, como se muestra con la función model() y se utiliza el modelo con el dataset con 5 iteraciones o epochs:

Código 8. Estructurando el modelo 3.

```
def model():
    model = Sequential()
    model.add(Conv2D(80, (3,3), strides = (1, 1), activation = 'relu'))
    model.add(Conv2D(64, (3,3), strides = (1, 1), activation = 'relu', input_s
hape = (240, 320, 3)))
    model.add(MaxPool2D(pool_size = (2,2)))
    model.add(Conv2D(64, (3,3), strides = (1,1), activation = 'relu'))
    model.add(Dropout(0.25))
    model.add(Flatten())

    model.add(Dense(128, activation = 'relu'))
    model.add(Dropout(0.5))
    model.add(Dense(4, activation = 'softmax'))

    model.compile(loss = 'categorical_crossentropy', optimizer = 'adadelta', m
etrics = ['accuracy'])

    return model

nn = model()
nn.fit_generator(dataset, steps_per_epoch = None, epochs = 5, verbose = 1)
nn.summary()
nn.save('Model.h5')
```

Al imprimir el resumen de la arquitectura y los parámetros entrenar con el comando model.summary(), se muetra en la figura 39.



Layer (type)	Output Shape	Param #
conv2d (Conv2D)	multiple	2240
conv2d_1 (Conv2D)	multiple	46144
max_pooling2d (MaxPooling2D)	multiple	0
conv2d_2 (Conv2D)	multiple	36928
dropout (Dropout)	multiple	0
flatten (Flatten)	multiple	0
dense (Dense)	multiple	148242560
dropout_1 (Dropout)	multiple	0
dense_1 (Dense)	multiple	516

Total params: 148,328,388
Trainable params: 148,328,388
Non-trainable params: 0

Figura 34. Resumen de la arquitectura del modelo 3 y los parámetros a entrenar. Fuente: el autor.

Al entrenar el modelo no se observan resultados apropiados ya que la métrica *accuracy* no supera 0.25 como ya se explicó.

Se realizaron también otros modelos que no fueron exitosos y no justifica mostrarlos aquí. Se muestra más bien el modelo que sí es exitoso.

Luego se ajustaron algunos parámetros y capas llegando a una verdadera clasificación como se muestra en las figuras siguientes. Primero se cargan las librerías necesarias de keras y luego se define la arquitectura de la red con las diferentes capas, con la última capa con 4 neuronas ya que se clasifican los cuatro tipos de leucocitos.

Código 9. Modelo 1 clasificador.

```
import os
import keras_preprocessing
from keras_preprocessing import image
from keras_preprocessing.image import ImageDataGenerator

# Modelo No. 1
TRAINING_DIR = "/content/drive/My Drive/Colab Notebooks/images/train/"
training_datagen = ImageDataGenerator(
      rescale = 1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')

VALIDATION_DIR = "/content/drive/My Drive/Colab Notebooks/images/test/"
validation_datagen = ImageDataGenerator(rescale = 1./255)

train_generator = training_datagen.flow_from_directory(
    TRAINING_DIR,
    target_size=(120,160),
    class_mode='categorical'
)
```

```
26. 
27. validation_generator = validation_datagen.flow_from_directory(
28.     VALIDATION_DIR,
29.     target_size=(120,160),
30.     class_mode='categorical'
31. )
32. 
33. model = tf.keras.models.Sequential([
34.     # Obserque que el input shape corersponde al tamaño deseado de la imagen
35.     # 240x320 Alto*ancho con 3 bytes por el color
36.     # Este es la primera convolucion
37.     tf.keras.layers.Conv2D(64, (3,3), activation='relu', input_shape=(120,160,
    3)),
38.     tf.keras.layers.MaxPooling2D(2, 2),
39.     # La segunda convolucion
40.     tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
41.     tf.keras.layers.MaxPooling2D(2,2),
42.     # La tercera convolucion
43.     tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
44.     tf.keras.layers.MaxPooling2D(2,2),
45.     # La cuarta convolucion
46.     tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
47.     tf.keras.layers.MaxPooling2D(2,2),
48.     # Flatten los resultados para alimentar una DNN
49.     tf.keras.layers.Flatten(),
50.     #tf.keras.layers.Dropout(0.5),
51.     # 512 neuron hidden layer
52.     tf.keras.layers.Dense(512, activation='relu'),
53.     tf.keras.layers.Dense(4, activation='softmax')
54. ])
55. 
56. model.summary()
57. model.compile(loss = 'categorical_crossentropy', optimizer='rmsprop', metrics=
    ['accuracy'])
58. history = model.fit_generator(train_generator, epochs=25, validation_data = va
    lidation_generator, verbose = 1)
59. saved_model_path = "/content/drive/My Drive/Colab Notebooks"
60. tf.saved_model.save(model, saved_model_path)
```

Se observa la estructura detallada del modelo resultado de la función model.summary():

```
Found 10381 images belonging to 4 classes.
Found 2511 images belonging to 4 classes.
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 118, 158, 64)      1792
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 59, 79, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 57, 77, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 28, 38, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 26, 36, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 13, 18, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 11, 16, 128)       147584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 5, 8, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 5120)              0
_________________________________________________________________
dense (Dense)                (None, 512)               2621952
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 2052
=================================================================
Total params: 2,884,164
Trainable params: 2,884,164
Non-trainable params: 0
```

Figura 35. Resumen del modelo 1. Fuente: el autor.

Se observa que hay 2´884.164 parámetros para entrenar en el modelo. En el entrenamiento, se observa que hay un aumento consistente en la métrica *accuracy* sobre la parte del dataset para el entrenamiento (imágenes de la carpeta *train*), no obstante, la métrica *val_accuracy* tiene altibajos. En la figura 44 se muestran las iteraciones primeras y últimas del total de 25:

```
_________________________________________________________________
Epoch 1/25
325/325 [==============================] - 6962s 21s/step - loss: 1.4039 - accuracy: 0.2585 - val_loss: 1.3820 - val_accuracy: 0.2836
Epoch 2/25
325/325 [==============================] - 100s 308ms/step - loss: 1.3431 - accuracy: 0.3504 - val_loss: 1.2584 - val_accuracy: 0.4110
Epoch 3/25
325/325 [==============================] - 99s 306ms/step - loss: 1.1663 - accuracy: 0.4810 - val_loss: 0.8465 - val_accuracy: 0.6432
Epoch 4/25
325/325 [==============================] - 99s 304ms/step - loss: 0.9179 - accuracy: 0.6062 - val_loss: 1.0752 - val_accuracy: 0.5842
Epoch 5/25
325/325 [==============================] - 98s 301ms/step - loss: 0.7796 - accuracy: 0.6692 - val_loss: 0.5450 - val_accuracy: 0.7471
...
Epoch 22/25
325/325 [==============================] - 97s 298ms/step - loss: 0.3913 - accuracy: 0.8563 - val_loss: 0.7119 - val_accuracy: 0.7308
Epoch 23/25
325/325 [==============================] - 98s 300ms/step - loss: 0.3536 - accuracy: 0.8651 - val_loss: 0.4643 - val_accuracy: 0.8519
Epoch 24/25
325/325 [==============================] - 96s 296ms/step - loss: 0.3824 - accuracy: 0.8605 - val_loss: 0.3508 - val_accuracy: 0.8570
Epoch 25/25
325/325 [==============================] - 97s 298ms/step - loss: 0.3977 - accuracy: 0.8623 - val_loss: 0.4548 - val_accuracy: 0.8582
WARNING:tensorflow:From /tensorflow-2.0.0/python3.6/tensorflow_core/python/ops/resource_variable_ops.py:1781: calling BaseResourceVari
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
INFO:tensorflow:Assets written to: /content/drive/My Drive/Colab Notebooks/assets
```

Figura 36. Entrenamiento modelo 1 que resultó exitoso. Fuente: el autor.

Este primer modelo se realizó con Tensorflow 2.0.0 y configurando el servidor de Google Colab con GPU.

Posteriormente se realizó un entrenamiento con Tensorflow versión `'2.1.0-rc1'` y configurando el servidor de Google colab con TPU, y se aumentó el número de iteraciones a epochs=50.

En este caso la métrica *accuracy* se rezaga con respecto al mismo entrenamiento del modelo en el caso anterior con 25 epochs y luego se estanca sin presentarse una mejoría significativa en el indicador *accuracy* e inclusive una desmejora en la última iteración del indicador *val_accuracy.*

Para generar una gráfica donde se observen las métricas de desempeño sobre la parte del dataset de entrenamiento (*train*), es decir *accuracy*, así como sobre las imágenes de validación o *val_accuracy*, se usa el siguiente código:

Código10. Graficación de las métricas del proceso de entrenamiento.

```
import matplotlib.pyplot as plt
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(len(acc))
plt.plot(epochs, acc, 'r', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend(loc=0)
plt.figure()
plt.show()
```

La gráfica generada de estos indicadores es la siguiente:

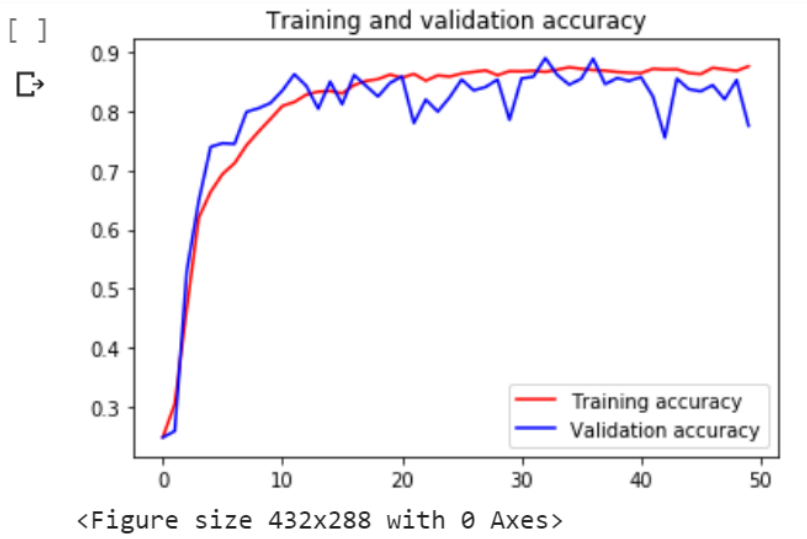


Figura 37. Precisiones del proceso de entrenamiento del modelo propio con epochs=50 y tensorflow 2.1.0. Fuente: el autor.

Donde se observa claramente que hasta la iteración correspondiente a epoch=25, no se presenta un incremento significativo de las precisiones.

Modelos Pre - entrenados. Se probaron varias redes pre - entrenadas, encontrándose que es necesario excluir la última capa de la red que se utilice con el comando include_top dejando este en False, y luego ajustando con la primera capa que se agrega para entrenarla con las nuevas imágenes.

Se utilizó una red *Inception* y aparentemente se logró un clasificador con un accuracy del 0.96 como se puede constatar en el notebook, después de alrededor de 11 horas de entrenamiento. Pero al probar la red entrenada con varias imágenes de las cuatro categorías alimentadas desde la carpeta *test*, se observó que lamentablemente las clasificaba a todas como *eosinófilos*. Luego se descartó este modelo.

Otra red que se entrenó fue con VGG16. Aquí también es fundamental la forma como se alimenta la imagen con el comando input_shape. En el siguiente código se observa la configuración para utilizar la red VGG16:

Código 11. Red entrenada con VGG16 y con Tensorfow 2.1

```
# Modelo No. 1k
TRAINING_DIR = "/content/drive/My Drive/Colab Notebooks/images/train/"
training_datagen = ImageDataGenerator(
      rescale = 1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')

VALIDATION_DIR = "/content/drive/My Drive/Colab Notebooks/images/test/"
validation_datagen = ImageDataGenerator(rescale = 1./255)

train_generator = training_datagen.flow_from_directory(
    TRAINING_DIR,
    target_size=(150,150),
    class_mode='categorical'
)

validation_generator = validation_datagen.flow_from_directory(
    VALIDATION_DIR,
    target_size=(150,150),
    class_mode='categorical'
)

conv_base=tf.keras.applications.VGG16(weights='imagenet',include_top=False,input_shape=(150, 150, 3))
conv_base.trainable = False
#conv_base.summary()

model1k = tf.keras.models.Sequential([
  conv_base,
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(256, activation='relu'),
  tf.keras.layers.Dense(4, activation='softmax')

])

model1k.compile(loss = 'categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
#Para tensorflow 3.1 usar model.fit()
history = model1k.fit(train_generator, epochs=14, validation_data = validation_generator, verbose = 1)

saved_model_path = "/content/drive/My Drive/Colab Notebooks/modelo1m"
tf.saved_model.save(model1k, saved_model_path)
```

Observe cómo para poder utilizar esta red, fue necesario cambiar la forma de las imágenes con el comando input_shape definiendo el tamaño a 150x150x3 en vez de 120x160x3. Esto lo hace el modelo de forma automática, sin requerir ajustar manualmente el tamaño de las imágenes. No obstante, cuando se va a utilizar el modelo ya entrenado de esta forma, la imagen que se alimente sí debe tener estas dimensiones de 150x150x3.

Adicionalmente, ya que se utilizó la última versión de Tensorflow, fue necesario cambiar la función *model.fit_generator()* a *model.fit()* (Google, s.f.).

En la siguiente figura, se puede observar la estructura de VGG16:

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 150, 150, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 150, 150, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 150, 150, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 75, 75, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 75, 75, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 75, 75, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 37, 37, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 37, 37, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 37, 37, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 37, 37, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 18, 18, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 18, 18, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 9, 9, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 4, 4, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
```

Figura 38. Arquitectura VGG16. Fuente: el autor.

Se observa que es una red relativamente pequeña en comparación a Inception y otras de mayor complejidad, sin embargo el número de parámetros para entrenar se eleva a 14.714.688

Se realiza el entrenamiento. En la figura 39 se observa la métrica *accuracy* durante este proceso.

Se observan menores resultados que con la red entrenada desde cero, esto se puede presentar por la forma de las imágenes de las células de la sangre cuyas características pueden ser muy diferentes a las imágenes con que se entrenó VGG16. No justifica volver a entrenar este modelo aumentando el número de iteraciones o epochs ya que se observa que si bien la métrica de accuracy presenta un aumento, la de val_accuracy se estanca en alrededor de 0.6.

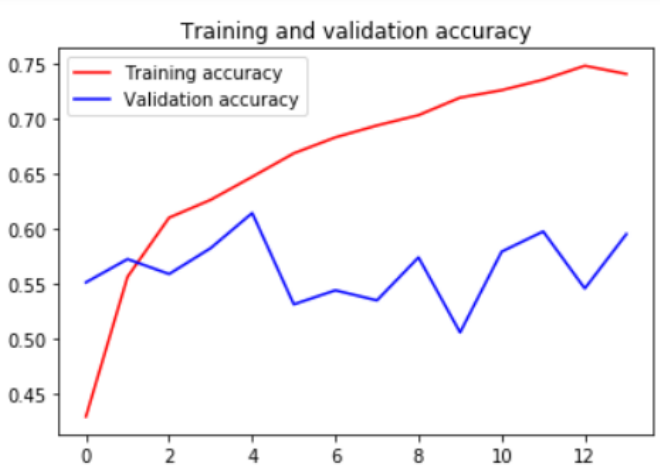


Figura 39. Gráfico de las métricas accuracy y val_ccuracy del entrenamiento de VGG16. Fuente: el autor.

Al utilizar el modelo que se entrenó desde cero que se considera el mejor y se toma como modelo base y producto final de este TFM, este se abre desde el notebook donde se encuentra el Detector con:

Código 12. Cargar el modelo 1 entrenado desde cero para su uso:

```
#se carga el arhivo de la ruta :'/content/drive/My Drive/Colab Notebooks/saved_model.pb'

new_model = tf.keras.models.load_model('/content/drive/My Drive/Colab Notebooks/')
```

Ahora se va a probar la métrica *accuracy*, para lo cual se utiliza el siguiente código que permite ingresar varias imágenes a la vez seleccionándolas desde el disco duro del equipo.

Código 13. Ingreso de imágenes desde cualquier fuente con el modelo base:

```
#Cargar varias imagenes para clasificarlas desde Disco duro, segun:
# - eosinophil: [1.0.0.0]
# - lymphocyte: [0.1.0.0]
# - monocyte:   [0.0.1.0]
# - neutrophil: [0.0.0.1]
import numpy as np
from google.colab import files
from keras.preprocessing import image

```

```
uploaded = files.upload()

for fn in uploaded.keys():

  # predicting images
  path = fn
  img = image.load_img(path, target_size=(120, 160))
  x = image.img_to_array(img)
  x = np.expand_dims(x, axis=0)

  images = np.vstack([x])
  classes = new_model.predict(images, batch_size=10)
  print(fn)
  print(classes)
  x=new_model.predict_classes(images, batch_size=10)
  print(x)
  if(x==3):
    print('neutrophil')
  elif(x==2):
    print('monocyte')
  elif(x==1):
    print('lymphocite')
  else:
    print('eosinophil')
```

Al ejecutar el código anterior, aparece el botón “Elegir archivos”, y al ingresar 7 imágenes al azar de la carpeta *train* y dentro de esta de la carpeta *neutrophil*, se observan los 7 aciertos.

Al seleccionar otras 7 imágenes de neutrófilos, acierta en 6. En total se seleccionaron 40 imágenes de neutrófilos y el modelo acertó en 37, lo cual da un *accuracy* de 0.93, que confirma el valor de 0.85 que se indicó durante el entrenamiento.

Luego se realizó lo propio con los monocitos, de 42 se tuvieron 35 aciertos: para un *accuracy* de 0.83

Se seleccionaron luego 42 imágenes de linfocitos y se presentaron 36 aciertos para un *acuracy* de 0.857.

Con las imágenes de eosinófilos se tuvo un acierto bajo ya que de 40 imágenes solo se tuvieron 26 aciertos para un *accuracy* de 0.65

Es decir, de 164 imágenes seleccionadas aleatoriamente se tuvieron 134 aciertos para un *accuracy* aproximado de 0.78, menor al reportado en el entrenamiento de 0.8623, no obstante, por ser una muestra se considera que no representa una diferencia significativa. La función *BinomCI* en el paquete *DescTools* en el software estadístico R, tiene varios métodos para calcular los intervalos de confianza para una proporción binomial. Según la documentación de R, la función se define (https://www.rdocumentation.org/, s.f.):

```
BinomCI(x, n, conf.level = 0.95, sides = c("two.sided", "left",
"right"),method = c("wilson", "wald", "agresti-coull", "jeffreys",
"modified wilson", "wilsoncc","modified jeffreys",
"clopper-pearson", "arcsine", "logit", "witting", "pratt"),rand = 123)
```

Aplicando esta función con n=164 y x=125 éxitos, se tiene:

Código 14. Establecer un Intervalo de confianza en el programa R

```
1.  library(DescTools)
2.  library(PropCIs)
3.  BinomCI(134, 164, conf.level = 0.95,method = "clopper-pearson")
```

Arrojando el intervalo de confianza:

```
> BinomCI(134, 164, conf.level = 0.95,method = "clopper-pearson")
           est    lwr.ci    upr.ci
[1,] 0.8170732 0.7492756 0.8730456
```

Este intervalo incluye el valor de la métrica *accuracy* dada por el modelo, es decir, con un nivel de confianza de 0.95 el *accuracy* del modelo es 0.8623

Ahora se realiza el mismo procedimiento con las imágenes de la carpeta *test*, que da:

-Neutrófilos: de 42 imágenes se tienen 35 aciertos para una accuracy de 0.83
-Monocitos: de 44 imágenes se tienen 36 aciertos para una accuracy de 0.82
-Linfocitos: 45 se tuvieron 40 aciertos para una accuracy de 0.89.
-Eosinófilos: de 42 imágenes se tuvieron 28 aciertos para una accuracy de 0.64

En total de 173 imágenes de prueba seleccionadas al azar se tuvieron 139 aciertos para un val_*accuracy* de 0.79, que no es una diferencia significativa con el valor dado por el modelo de 0.8582. Al ejecutar la función BinomCI en R para este caso, se obtiene:

```
> BinomCI(139, 173, conf.level = 0.95,method = "clopper-pearson")
           est    lwr.ci    upr.ci
[1,] 0.8034682 0.7363464 0.8598897
```

Se observa que el intervalo de confianza incluye el val_*accuracy* de 0.8582. Luego a un nivel de confianza de 0.95, se concluye que el val_*accuracy* es de 0.8582.

3.3.2 Entrenamiento del Detector

Para el desarrollo del modelo Detector, se usó una red YOLO y las librerías de ImageAI así como un modelo preentrenado facilitado por los autores de ImageAI, utilizando el dataset con las etiquetas tanto de bounding boxes como de clasificación para diferenciar entre RBCs, WBCs y plaquetas.

Se utiliza la librería especializada en detección de objetos ImageAI que se instala con facilidad y en poco tiempo.

Se utilizó una red pre entrenada por los mismos autores de ImageAI (Olafenwa):

Código 15. Entrenamiento de la red para el Detector

```
from imageai.Detection.Custom import DetectionModelTrainer

trainer = DetectionModelTrainer()
trainer.setModelTypeAsYOLOv3()
trainer.setDataDirectory(data_directory="/content/drive/My Drive/Colab Notebooks/Blood-Cell-Count-Dataset2/")
trainer.evaluateModel(model_path="/content/drive/My Drive/Colab Notebooks/Blood-Cell-Count-Dataset2/models",
                      json_path="drive/My Drive/Colab Notebooks/Blood-Cell-Count-Dataset2/json/detection_config.json",
                      iou_threshold=0.5, object_threshold=0.3, nms_threshold=0.5)
```

El entrenamiento toma alrededor de 11 horas.

3.3.3 Ensamble del Detector - Clasificador

Se procede ahora a combinar los modelos obtenidos del Detector - Clasificador de los tipos principales de leucocitos. Ya que el Clasificador se tiene disponible, se ensamblan el Detector y luego el Clasificador. Para esto fue necesario que la imagen que se pasa por el detector de tamaño width: 640 height: 480 y en que se diferencian los tres tipos de células principales entre RBCs, WBCs y plaquetas. En la figura 59 se observa un esquema de funcionamiento del sistema desarrollado, en que se tiene una imagen de entrada al Detector y aquí se entrega una imagen de salida con los bounding box y etiquetas de las células detectadas en la imagen. Luego se pasa esta a la función de recorte de los leucocitos o WBC y la imagen resultante se ingresa al clasificador en el cual se da como salida el tipo de leucocito.

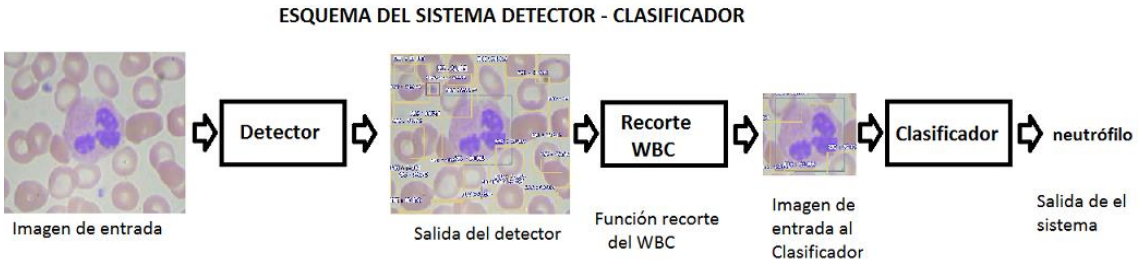


Figura 40. Esquema del sistema Detector – Clasificador. Fuente: el autor.

Se tiene un notebook llamado Deteccion_Clasificacion_WCZ.ipynb, en el cual se puede correr el sistema completo. Inicialmente se realizan los pasos indicados en el primer bloque de texto del notebook como se indica en el siguiente recuadro:

Sistema Detección - Clasificación de leucocitos.

Ejecuta las celdas necesarias para cargar todos los modelos, librerías y funciones:

1. Conectarse al Google - Drive
2. Importar Tensorflow y Ver la versión
3. Cargar el modelo Clasificación de leeucocitos
4. Ejecutar la función imrecortada()
5. Instalar la libreria ImageAI
6. Cargar el modelo Detección Células normales sangre.
7. Proceder con los pasos para ejecutar los modelos.

Para cada uno de los pasos indicados se tiene una celda en el notebook. El paso 7, se especifica en el mismo notebook como se indica en el siguiente recuadro:

Ejecutar Sistema

Ya cargados todos los modelos, librerías y funciones, proceder con el sistema Detección - Clasificación de leucocitos.

1. Ingrese la ruta de la imagen de Entrada al sistema
2. Ingrese la ruta de la imagen a Guardar.
3. Después puede Ejecutar Celdas siguientes con Ctrl+F10

Los pasos 1 y 2 requieren indicar la ruta para cargar la imagen de entrada al Detector y así mismo la ruta y nombre con que se va a guardar a imagen de salida del detector, es decir, con los objetos detectados con sus etiquetas y bounding boxes correspondientes.

La salida del Detector da un diccionario como el indicado en la figura 44.

```
RBC  :  83.80960822105408  :  [13, 28, 139, 144]
RBC  :  94.98987793922424  :  [454, 69, 566, 173]
WBC  :  99.37371015548706  :  [197, 123, 433, 333]
RBC  :  90.60249924659729  :  [143, 334, 261, 441]
RBC  :  97.21892476081848  :  [26, 377, 154, 475]
RBC  :  94.58165168762207  :  [404, 0, 523, 70]
RBC  :  92.47880578041077  :  [91, 0, 190, 87]
RBC  :  92.0530378818512  :  [202, 1, 301, 92]
RBC  :  98.20033311843872  :  [517, 8, 641, 85]
RBC  :  95.16353607177734  :  [309, 30, 413, 139]
RBC  :  68.46207976341248  :  [163, 59, 288, 191]
RBC  :  90.26426076889038  :  [70, 194, 181, 313]
RBC  :  79.1100025177002  :  [5, 217, 118, 348]
RBC  :  72.81860113143921  :  [501, 258, 612, 356]
RBC  :  89.90716934204102  :  [375, 278, 478, 366]
RBC  :  88.87978792190552  :  [228, 328, 351, 405]
RBC  :  92.54742860794067  :  [535, 315, 638, 403]
RBC  :  77.37593054771423  :  [374, 383, 482, 469]
RBC  :  68.54408383369446  :  [256, 436, 360, 480]
RBC  :  98.80066514015198  :  [480, 418, 596, 481]
RBC  :  82.8016459941864  :  [0, 1, 85, 57]
Platelets  :  97.5511908531189  :  [128, 85, 175, 122]
RBC  :  58.64359140396118  :  [0, 114, 12, 208]
```

Figura 41. Carga del modelo entrenado y ejecución con la imagen BloodImage_0039. Fuente: el autor.

Se observa de la corrida que se detectan los tres tipos de células principales normales de la sangre. Luego para ejecutar el clasificador se toman las coordenadas de los leucocitos (WBC) correspondientes que en este caso corresponde a la tercera célula detectada, este procedimiento se realiza automáticamente tomando la información del diccionario de Python generado en la detección. Previo a la alimentación de la imagen al Clasificador, se debe recortar la imagen alrededor del leucocito detectado. Se observa en la figura 45 que sólo hay un leucocito el cual aparece con manchas azules.

Se muestra la imagen original en la figura 45 así como la fracción del archivo xml de la carpeta Annotations correspondiente en que se encuentran los datos del WBC o leucocito de interés (en el anexo se encuentra el archivo completo):

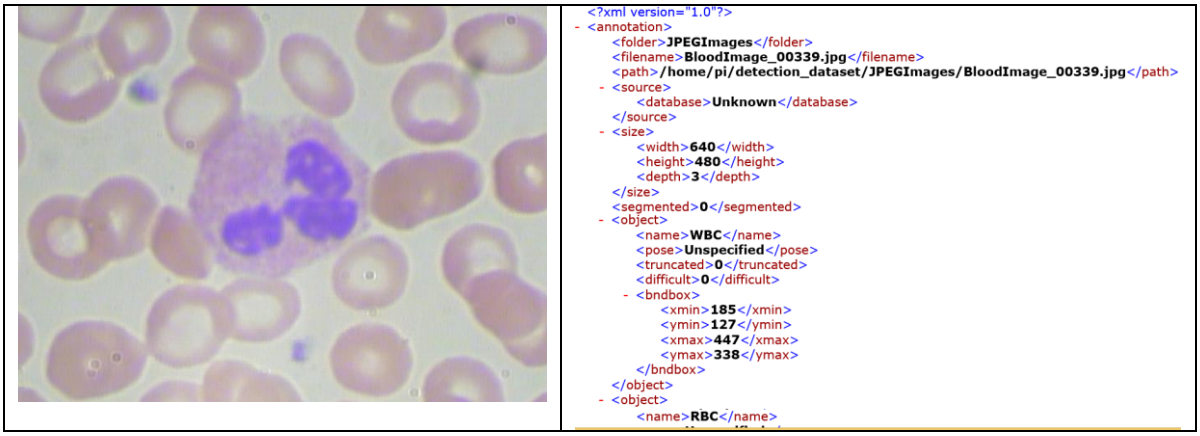


Figura 42. Imagen original y una muestra del archivo xml correspondiente. Fuente: el autor.

Del archivo xml, se encuentran la información del bounding box para el leucocito como:

<bndbox>
<xmin>185</xmin>
<ymin>127</ymin>
<xmax>447</xmax>
<ymax>338</ymax>
</bndbox>

Al ejecutar el Detector da la siguiente información para el leucocito detectado:

```
WBC  :  99.37371015548706  :  [197, 123, 433, 333]
```

Luego se observan unas diferencias pequeñas, que se cuantifican tomando como el valor verdadero el dado por el archivo xml, y calculando el error relativo porcentual, según:

$$x_{min} = \frac{185 - 197}{185} x100 = -6.5\%$$

$$x_{max} = \frac{447 - 433}{447} x100 = -3.3\%$$

$$y_{min} = \frac{127 - 123}{127} x100 = 3.2\%$$

$$y_{max} = \frac{338 - 333}{338} x100 = 1.5\%$$

Se observa que la precisión es alta ya que se presentan errores muy bajos.

Para realizar el recorte del leucocito detectado de la imagen de entrada, el autor de este TFM desarrolló la función:

Código 16. Definición de la función imrecortada()

```
#Funcion que ajusta la imagen recortada dadas las coordenadas
#xmin,ymin,xmax,ymax, a un tamaño de 320x240
#Programada por Wilson Castro Z.
def imrecortada(xmin,ymin,xmax,ymax,width,height):
  ancho=xmax-xmin
  alto=ymin-ymax
  while(ancho<320):
    if(xmin-1>=0):
      xmin=xmin-1
    xmaximo=width-xmin
    if(xmax+1<=xmaximo):
      xmax=xmax+1
    ancho=xmax-xmin
  while(alto<240):
    if(ymin-1>=0):
      ymin=ymin-1
    ymaximo=height-ymin
    if(ymax+1<ymaximo):
       ymax=ymax+1
    alto=ymax-ymin
  return xmin,ymin,xmax,ymax
```

En esta función, específicamente desarrollada por el autor de este TFM, se requieren los parámetros indicados dados por el Detector y con el ancho y alto conocidos de la imagen original alimentada a la red, que en este caso son de tamaño 640 x 480. Esta función toma estas coordenadas directamente del diccionario de python así como el tamaño de la imagen para recortar una fracción de la misma que incluya la parte del leucocito. La función centra la imagen del leucocito a recortar y amplía esta zona hasta obtener un tamaño de ancho 320 por 240 de alto, con lo cual se obtiene la imagen recortada de la imagen 1B.

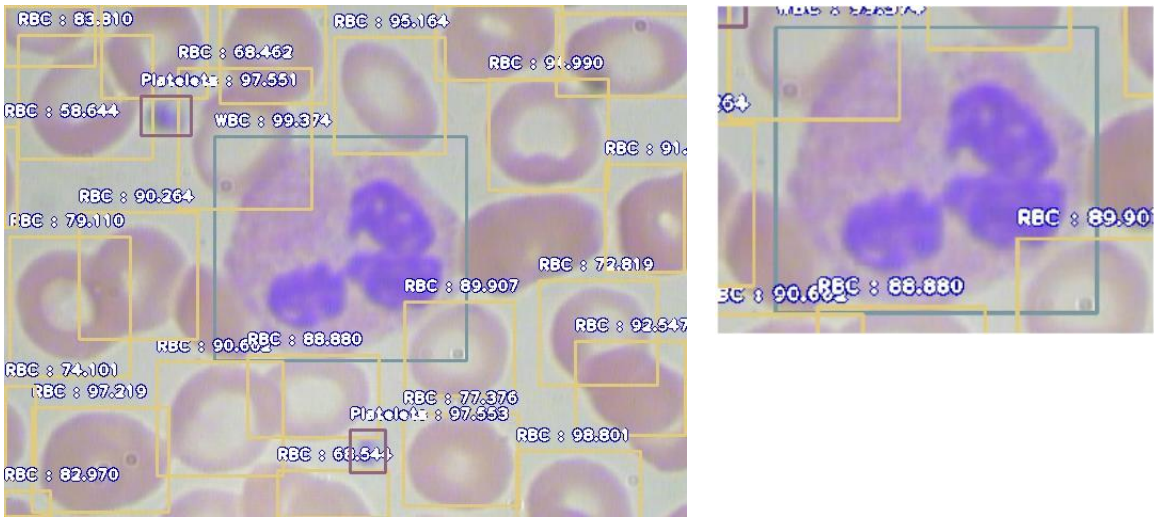


Figura 43. Imagen resultante al aplicar el Detector (izquierda) y luego a esta se le aplica la función imrecortada() (derecha). Fuente: el autor.

Luego es necesario reducir el tamaño de la imagen a la mitad manteniendo la proporción para llegar a un tamaño de 160X120 (ancho x alto), con el código:

Código 17. Reducción de la imagen de salida del Detector.

```
#Reducir la imagen a 120x160 (la mitad para correrla al clasificador)
# Downsize the image with an ANTIALIAS filter (gives the highest quality)
im1 = im1.resize((160,120),Image.ANTIALIAS)
#Show the new image
plt.imshow(im1)
plt.axis('Off')
plt.show()
```

Se ejecuta el modelo automáticamente que toma la imagen recortada y reducida de tamaño con el siguiente código en el cual se ha programado la forma de interpretar el vector al tipo de leucocito correspondiente, a diferencia del proceso anterior de prueba de Clasificador con varias imágenes que se podían cargar del disco duro o cualquier origen, aquí se carga sólo una imagen que es la resultante de los procesos de detección seguido del recorte al invocar la función imrecortada():

Código 18. Alimentación automática de la imagen al Clasificador.

```
#Cargar la imagen recortada para clasificarla segun:
# - eosinophil: [1.0.0.0.]
# - lymphocyte: [0.1.0.0.]
# - monocyte:   [0.0.1.0.]
# - neutrophil: [0.0.0.1.]
import numpy as np
from google.colab import files
from keras.preprocessing import image

x = image.img_to_array(im1)
x = np.expand_dims(x, axis=0)

images = np.vstack([x])
```

```
classes = new_model.predict(images, batch_size=10)

print(classes)
x=new_model.predict_classes(images, batch_size=10)
print(x)
if(x==3):
  print('neutrophil')
elif(x==2):
  print('monocyte')
elif(x==1):
  print('lymphocite')
else:
  print('eosinophil')
```

Y el resultado es el esperado:

```
[[0. 0. 0. 1.]]
[3]
neutrophil
```

Que efectivamente es un neutrófilo porque es un leucocito con núcleo multilobulado (tres núcleos).

3.4 Paso 4. Evaluar el modelo

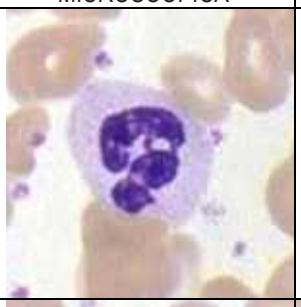
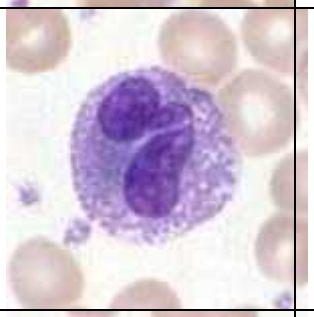
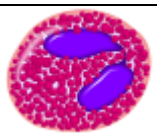
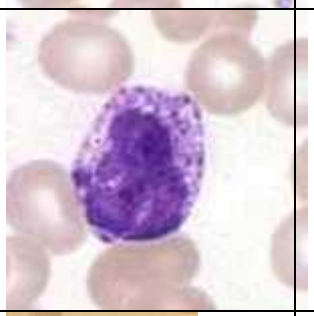
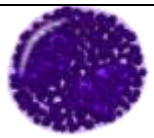
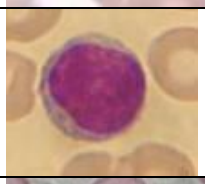
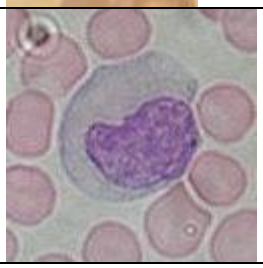
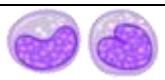
Por último, es necesario evaluar la red entrenada. Para cada una de las imágenes en el conjunto de pruebas, se le presentan a la red y se le pide a esta que prediga la etiqueta de la imagen. Luego se tabulan las predicciones del modelo para una imagen en el conjunto de prueba.

Finalmente, estas predicciones del modelo se comparan con las etiquetas básicas verdaderas (*ground-truth* labels) del conjunto de pruebas.

Las etiquetas básicas verdaderas representan cuál es realmente la categoría de imagen. A partir de ahí, es posible calcular el número de predicciones que el clasificador obtuvo correctamente y calcular informes agregados como la métrica *accuracy* que se utilizan para cuantificar el rendimiento de la red en su conjunto.

Para el modelo total: Detector seguido de clasificador, no se tienen las etiquetas del tipo de leucocito detectado. En este caso, es necesario realizar por inspección de acuerdo con las características de los leucocitos como se observa en la tabla 1 (wikipedia, s.f.).

Tabla 1. Características de los tipos de leucocitos

TIPO LEUCOCITO	APARIENCIA MICROSCÓPICA	DIAGRAMA	FORMA
Neutrófilos			Tienen un núcleo multilobulado que puede asemejar múltiples núcleos, de ahí se deriva el nombre leucocito polimorfonuclear.[9] El citoplasma puede parecer transparente debido a los gránulos que se tiñen color lila pálido.
Eosinófilos			En general, su núcleo es bi-lobulado. El citoplasma está lleno de gránulos que, con tinción de eosina, asumen un color anaranjado característico.
Basófilos			Su núcleo es bi- o tri-lobulado, pero es difícil de detectar ya que se oculta por el gran número de gránulos gruesos, estos gránulos son característicamente azules bajo la tinción HyE.
Linfocitos			Se distinguen por un núcleo que se tiñe fuertemente y cuya locación puede o no ser excéntrica, y por tener poco citoplasma.
Monocitos			Su núcleo tiene forma de riñón y no tienen gránulos y contienen abundante citoplasma.

Se realizan 20 corridas completas (Detector seguido de clasificador) para revisar el desempeño de todo el proceso (Ver anexo). Los resultados obtenidos son muy satisfactorios: el detector logró ubicar con alta precisión todos los leucocitos ya que las coordenadas dadas para el bounding box coincidieron con unas diferencias de menos de 5 pixeles en todos los casos al comparar

con las correspondientes coordenadas del archivo xml. El clasificador presentó el 90% de aciertos.

3.5 Generalización.

La Generalización es la capacidad de la red de predecir correctamente la clase de una imagen que no está en ninguno de los subconjuntos de entrenamiento o prueba (Rosebroock, 2017). Al utilizar imágenes de dos datasets, es decir, de dos fuentes diferentes para entrenar a red para el Clasificador, se puede lograr una mayor generalización.

4. Conclusiones

- Se ha desarrollado un sistema de Detección – Clasificación de los diferentes tipos de células normales de la sangre periférica, de alto desempeño, buena precisión y computacionalmente rápido. El sistema toma una imagen, detecta las células de la sangre y luego permite recortar los leucocitos y clasificarlos según el tipo.

- Se desarrolló un modelo con excelente desempeño para la detección de los tres tipos principales de células normales de la sangre, a saber, eritrocitos, leucocitos y plaquetas.

- Se entrenó un modelo propio para la clasificación de los leucocitos entre los tipos neutrófilos, linfocitos, monocitos y eosinófilos que presentó una exactitud de 0.8623 y una exactitud de validación de 0.8582.

- Se encontró un aumento notable en el desempeño del clasificador en el sistema ensamblado, porque al tener la imagen del leucocito ampliada al recibirla del proceso de detección, mejora el proceso de reconocimiento de las características de la imagen.

- Sobre la metodología, se tuvieron dificultades para implementar una red pre entrenada para el clasificador de las células. Los resultados con VGG16 tuvieron unas métricas muy bajas. Con otras redes como Inception, el entrenamiento no superó la clasificación dada por el azar del 25% para las cuatro categorías de células.

- El mayor obstáculo con el desarrollo de este proyecto fue la no disponibilidad de un dataset completo, sin embargo, una novedad de este trabajo es que se combinaron dos datasets de diferentes fuentes y se obtuvieron excelentes resultados. Adicionalmente, por la variedad del tipo de imágenes, se obtiene una mayor generalidad en la aplicación de los productos.

4.1 Perspectivas de desarrollos futuros

- Se pueden realizar líneas de trabajo en el futuro con las metodologías de este TFM, que incluya células de la sangre normales y otras con alguna patología. Este tipo de trabajo será de utilidad para diagnosticar la enfermedad correspondiente.

- Se pueden reentrenar los modelos con una dataset propio y mejorar el desempeño, con esto se podría llevar el Detector – Clasificador a una aplicación como en una página web.

4. Glosario

Acrónimos utilizados en este TFM.

AEs: auto encoders.
CNN: convolutional neural networks.
CPU: Central Processing Unit.
DBNs :deep belief networks
DL: Deep Learning
DNNs: deep neural networks.
E.A. :emergent architectures.
GPU: Graphics Processing Unit.
I.A. Inteligencia Artificial.
ML: Machine Learning
MLP: multilayer perceptron
RGB: Red Green Blue. RGB es un modelo de color de los 3 canales básicos.
- ReLU: rectified linear unit.
RNN: recurrent neural networks.
RBC: Red Blood Cell, glóbulo rojo o eritrocito.
SAE : stacked auto-encoder
SGD: stochastic gradient descent.
SVM: Support Vector Machine.
WBC: White Blood Cell, glóbulo blanco o leucocito.
VGG16: Visual Geometry Group, es una arquitectura de CNN con 16 capas.

5. Bibliografía

Chollet, F. (2018). *Deep Learning with Python.* USA: Manning Publiations Co.

Kaiming, H., Xiangyu, Z., & Shaoqing, R. (2018). Deep Residual Learning for Image Recognition. *Microsoft research @microsoft.com*.

Krizhevsky, A. (2012). *ImageNet Classification with Deep Convolutional Neural Networks.* Curran Associates.

Lecun, Y. (1986). *Learning Processes in an Asymmetric Threshold Network.* France: Springer-Verlag.

LeCun, Y., & Bengio, Y. (1995). Convolutional Networks for Images, Speech, and Time-Series, in Arbib, M. A. *The Handbook of Brain Theory and Neural Networks, MIT Press*.

Moolayil, J. (2019). *Learn Keras for Deep Neural Networks.* Apress.

Orkin, S., & Zon, L. (2008). *SnapShot: hematopoiesis.*

Rosebroock, A. (2017). *Deep Learning for Computer Vision with Python.* PYIMAGESEARCH.

Wang, Q., Sun, M., & Yang, S. (2019). Deep learning approach to peripheral leukocyte recognition. *PLoS ONE 14(6):e0218808. https://doi.org/10.1371/journal.pone.0218808*.

Young, I. (1972). The Classificaction of White Blood Cells. *IEEE Transactions on Biomedical Engineering, Vol 4*, 291 - 298.

6. Anexos

Cuenta github: github/alfredwilson/TFM

CORRIDAS COMPLETAS CON EL DETECTOR – CLASIFICADOR

Se toman las coordenadas del bounding box dadas por el Detector de los leucocitos (WBC) en cada imagen de entrada de la carpeta test y se comparan en paralelo con las coordenadas correspondientes dadas por el archivo xml. Se indica si la detección fue satisfactoria. Luego se recorta la imagen con la función imrecortada() y se alimenta al modelo 1 de clasificación, también se indica si hubo acierto o no.

Imagen 0344

Resultado Detector

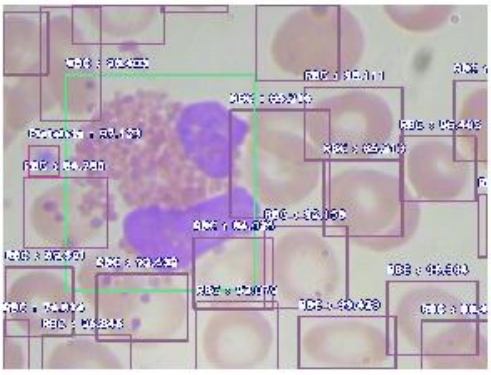
```
RBC  :  98.1108546257019  :  [398, 108, 527, 204]
RBC  :  85.216224193573  :  [300, 138, 431, 280]
WBC  :  98.40316772460938  :  [82, 92, 332, 376]
RBC  :  84.78481769561768  :  [27, 227, 138, 320]
RBC  :  92.63225197792053  :  [123, 354, 244, 442]
RBC  :  96.30359411239624  :  [507, 364, 626, 461]
RBC  :  69.57310438156128  :  [91, 1, 213, 51]
RBC  :  74.7537910938263  :  [334, 0, 443, 99]
RBC  :  85.21323204040527  :  [423, 204, 527, 304]
RBC  :  92.15754270553589  :  [345, 291, 454, 399]
RBC  :  84.02567505836487  :  [251, 306, 356, 400]
RBC  :  89.59082961082458  :  [3, 345, 92, 436]
RBC  :  92.07916259765625  :  [255, 391, 362, 480]
RBC  :  96.84617519378662  :  [51, 435, 168, 481]
RBC  :  99.42905306816101  :  [390, 410, 518, 479]
RBC  :  71.3475227355957  :  [209, 0, 296, 9]
RBC  :  60.12839674949646  :  [0, 9, 59, 92]
RBC  :  74.18230772018433  :  [49, 3, 128, 152]
RBC  :  75.83532333374023  :  [595, 99, 643, 206]
Platelets  :  97.12311625480652  :  [34, 185, 74, 225]
RBC  :  96.40308022499084  :  [524, 173, 624, 258]
RBC  :  86.81097030639648  :  [627, 249, 639, 332]
RBC  :  74.81399178504944  :  [0, 414, 34, 482]
RBC  :  88.43879103660583  :  [552, 416, 642, 480]
```

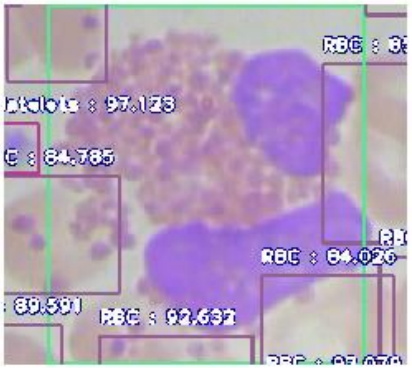
Bounding box del leucocito en el archivo xml

```
- <object>
      <name>WBC</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
    - <bndbox>
          <xmin>74</xmin>
          <ymin>108</ymin>
          <xmax>336</xmax>
          <ymax>359</ymax>
      </bndbox>
  </object>
```

```
WBC  :  98.40316772460938  :  [82, 92, 332, 376]
```

Y se observa que la detección es satisfactoria.
Se observan las imágenes dadas por el detector y de la misma, la imagen recortada del leucocito con la función imrecortada().






```
width: 640 height:  480
```

Finamente al alimentarla al Clasificador, se obtiene:

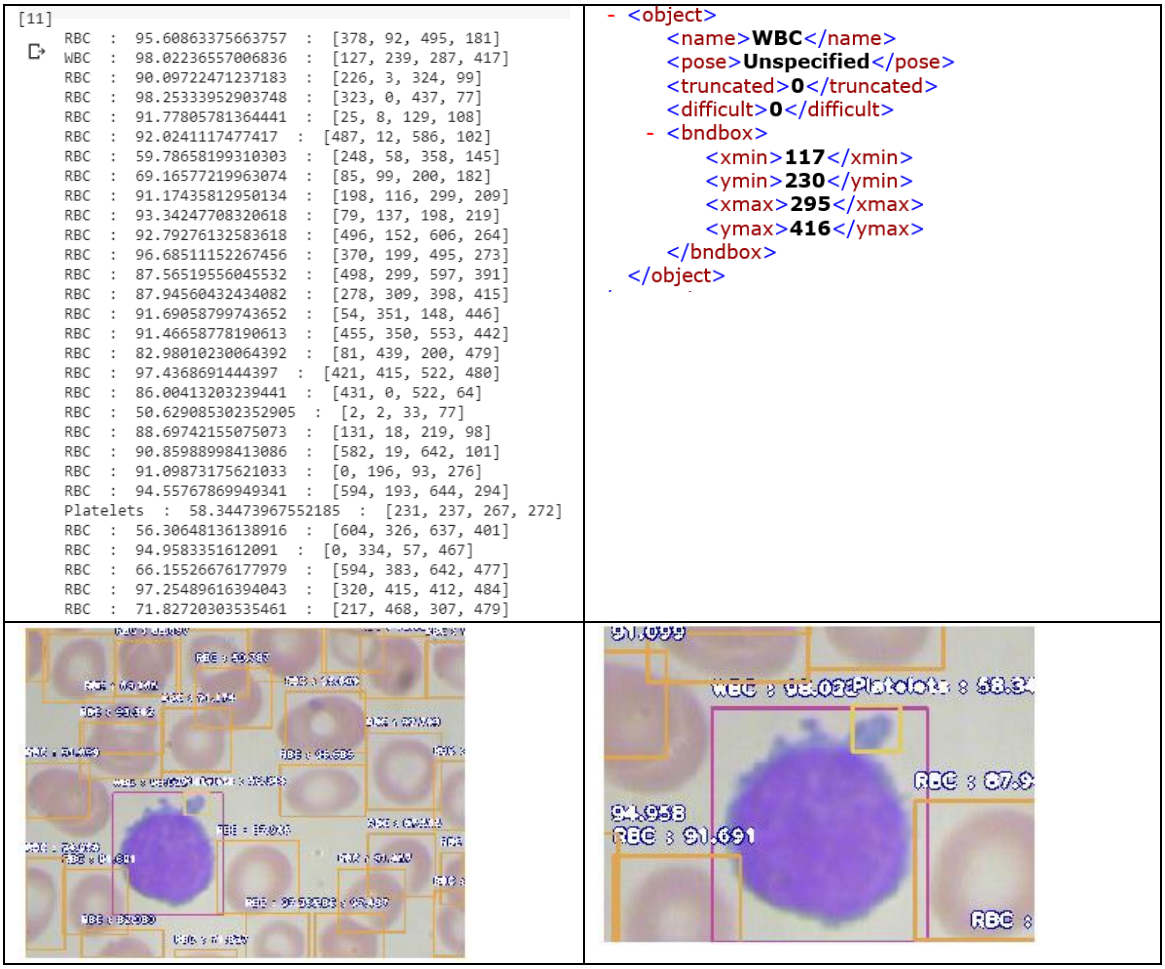
```
[[1. 0. 0. 0.]]
[0]
     eosinophil
```

El resultado es correcto. Se observa que el núcleo es bi-lobulado y el citoplasma está lleno de gránulos.

Se sigue un procedimiento similar, se muestran en una misma tabla los resultados del detector (izquierda superior), luego las coordenadas del bounding box correspondiente al leucocito según el archivo xml. La imagen generada por el detector se muestra en la parte inferior izquierda y la imagen recortada del leucocito al aplicar la función imrecortada() se muestra en la parte

inferior derecho. Esto para todas las 9 corridas de los modelos Detector-Clasificador.

Imagen 00356



WBC : 98.02236557006836 : [127, 239, 287, 417]

Resultado detector: satisfactorio.

Al ejecutar el clasificador arroja:

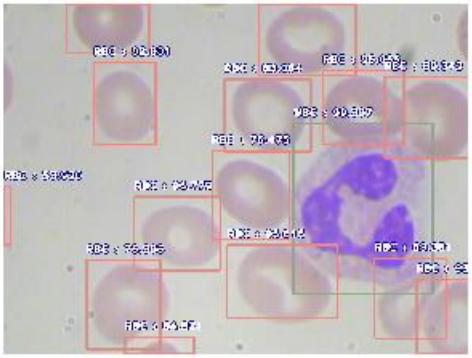
```
[[0. 1. 0. 0.]]
[1]
lymphocite
```

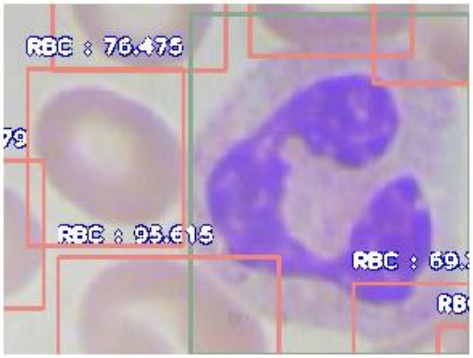
Resultado correcto.

Imagen 00384

```
RBC  :  96.57504558563232  :  [352, 1, 485, 98]
RBC  :  95.05522847175598  :  [440, 91, 551, 192]
RBC  :  83.06416869163513  :  [304, 103, 423, 202]
WBC  :  99.30726885795593  :  [399, 164, 591, 397]
RBC  :  96.77918553352356  :  [180, 264, 298, 365]
RBC  :  95.61543464660645  :  [308, 330, 460, 430]
RBC  :  98.30310940742493  :  [86, 1, 200, 67]
RBC  :  92.09136962890625  :  [124, 80, 209, 192]
RBC  :  89.34213519096375  :  [525, 100, 649, 211]
RBC  :  76.47548317909241  :  [287, 201, 394, 322]
RBC  :  69.38087344169617  :  [511, 348, 610, 458]
RBC  :  75.3058135509491  :  [114, 351, 216, 473]
RBC  :  53.92008423805237  :  [1, 250, 9, 330]
RBC  :  93.732750415802  :  [569, 377, 641, 478]
RBC  :  84.82100367546082  :  [167, 456, 262, 479]
```

```
<object>
    <name>WBC</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
  - <bndbox>
        <xmin>393</xmin>
        <ymin>175</ymin>
        <xmax>597</xmax>
        <ymax>391</ymax>
    </bndbox>
</object>
```






```
WBC  :  99.30726885795593  :  [399, 164, 591, 397]
```

Que es claramente un resultado satisfactorio.

Y el clasificador da

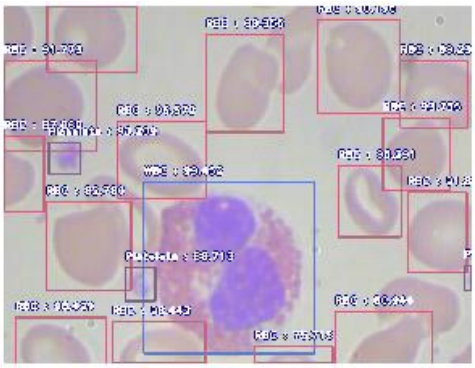
```
[[0. 0. 0. 1.]]
[3]
neutrophil
```

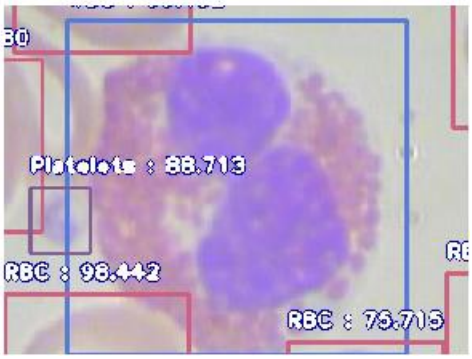
Resultado correcto, tiene un núcleo trilobulado.

Imagen 387

```
RBC  :  90.62964916229248  :  [59, 0, 182, 88]
RBC  :  91.77256226539612  :  [0, 73, 127, 195]
RBC  :  95.57196497917175  :  [155, 155, 276, 258]
WBC  :  99.45185780525208  :  [191, 236, 425, 466]
RBC  :  90.19623398780823  :  [429, 18, 542, 144]
RBC  :  83.35579633712769  :  [276, 38, 382, 169]
RBC  :  89.2381489276886  :  [542, 73, 634, 163]
RBC  :  93.75550150871277  :  [518, 150, 610, 247]
RBC  :  91.80420637130737  :  [552, 250, 633, 357]
RBC  :  82.78005719184875  :  [58, 263, 174, 381]
RBC  :  98.49905967712402  :  [16, 418, 140, 482]
RBC  :  98.44155311584473  :  [148, 424, 275, 479]
RBC  :  80.94366788864136  :  [453, 410, 586, 482]
Platelets  :  93.20899844169617  :  [61, 183, 105, 225]
RBC  :  87.06897497177124  :  [0, 175, 55, 276]
RBC  :  95.29095888137817  :  [457, 214, 543, 310]
Platelets  :  67.11605787277222  :  [629, 348, 639, 382]
Platelets  :  88.71265649795532  :  [166, 351, 207, 396]
RBC  :  75.71455836296082  :  [343, 457, 450, 479]
```

```
<object>
    <name>WBC</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
  - <bndbox>
        <xmin>202</xmin>
        <ymin>236</ymin>
        <xmax>430</xmax>
        <ymax>478</ymax>
    </bndbox>
</object>
```






```
WBC   :   99.45185780525208   :   [191, 236, 425, 466]
```

Resultado del detector: satisfactorio.

Clasificador:

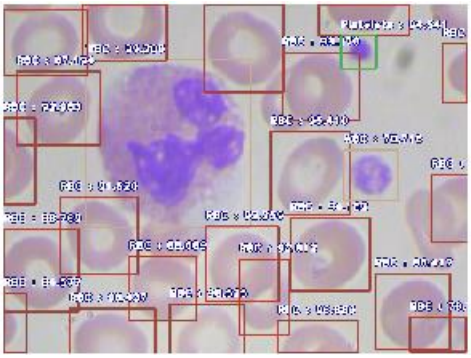
```
[[1. 0. 0. 0.]]
[0]
eosinophil
```

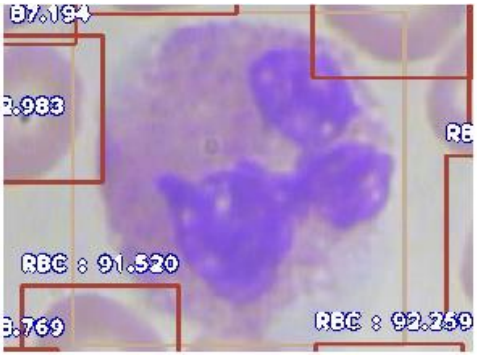
Resultado correcto, su núcleo es bilobulado.

Imagen 395

```
WBC   :   99.20000433921814   :   [113, 76, 340, 306]
RBC   :   94.15196776390076   :   [395, 293, 505, 394]
RBC   :   86.41650080680847   :   [512, 370, 615, 474]
RBC   :   69.51628923416138   :   [434, 1, 559, 42]
RBC   :   89.86836075782776   :   [0, 7, 115, 98]
RBC   :   94.34550404548645   :   [109, 1, 225, 77]
RBC   :   84.91932153701782   :   [277, 0, 386, 121]
RBC   :   93.27040314674377   :   [384, 66, 490, 159]
RBC   :   87.19382286071777   :   [18, 92, 133, 193]
RBC   :   95.40985822677612   :   [368, 175, 477, 289]
RBC   :   91.52045249938965   :   [78, 265, 185, 371]
RBC   :   88.76850008964539   :   [0, 309, 103, 422]
RBC   :   92.25901365280151   :   [279, 305, 379, 408]
RBC   :   88.6890709400177   :   [173, 346, 265, 435]
RBC   :   98.43709468841553   :   [91, 418, 209, 482]
RBC   :   98.5247015953064   :   [229, 412, 331, 481]
RBC   :   96.87953591346741   :   [378, 435, 489, 479]
Platelets   :   94.54118609428406   :   [465, 43, 514, 88]
RBC   :   77.07125544548035   :   [605, 54, 638, 135]
RBC   :   92.98291206359863   :   [0, 156, 44, 275]
WBC   :   70.71590423583984   :   [469, 200, 544, 269]
RBC   :   72.9562759399414   :   [590, 234, 645, 326]
RBC   :   92.8118646144867   :   [325, 350, 394, 454]
RBC   :   84.5767617225647   :   [0, 397, 31, 478]
RBC   :   70.70252299308777   :   [560, 430, 637, 477]
```

```
<object>
    <name>WBC</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
  - <bndbox>
        <xmin>114</xmin>
        <ymin>66</ymin>
        <xmax>351</xmax>
        <ymax>294</ymax>
    </bndbox>
</object>
```






```
WBC   :   99.20000433921814   :   [113, 76, 340, 306]

WBC   :   70.71590423583984   :   [469, 200, 544, 269]
```

Aquí se detectaron dos leucocitos. Se toma el primero que es más de un tamaño apropiado y el resultado es satisfactorio al compararar con el archivo xml. El segundo leucocito detectado no está descrito en el archivo xml.

Resultado clasificador:

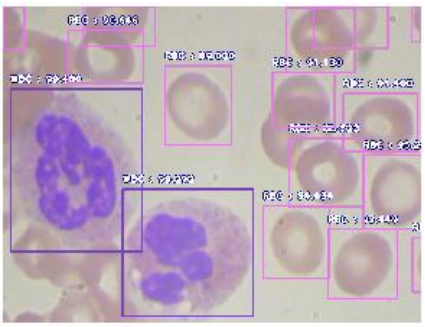
```
[[0. 0. 0. 1.]]
[3]
neutrophil
```

Resultado correcto. Tiene el núcleo trilobulado.

Imagen 374

```
WBC  :  99.48017001152039  :  [11, 125, 212, 372]
WBC  :  99.29207563400269  :  [183, 277, 383, 470]
RBC  :  91.61619544029236  :  [122, 0, 233, 60]
RBC  :  62.94776797294617  :  [475, 0, 589, 63]
RBC  :  87.30111122131348  :  [433, 2, 538, 100]
RBC  :  90.64646363258362  :  [100, 36, 214, 116]
RBC  :  89.02010917663574  :  [248, 90, 350, 210]
RBC  :  91.39967560768127  :  [520, 132, 634, 220]
RBC  :  86.52255535125732  :  [438, 200, 553, 303]
RBC  :  94.43421959877014  :  [398, 302, 499, 413]
RBC  :  92.44922995567322  :  [497, 338, 603, 443]
RBC  :  59.4738245010376  :  [625, 0, 639, 54]
RBC  :  87.48820424079895  :  [1, 0, 33, 65]
RBC  :  91.13032221794128  :  [412, 100, 507, 183]
RBC  :  95.24088501930237  :  [551, 225, 641, 337]
RBC  :  86.36489510536194  :  [626, 350, 640, 433]
```

```
<object>
    <name>WBC</name>
    <pose>Unspecified</pose>
    <truncated>1</truncated>
    <difficult>0</difficult>
  - <bndbox>
        <xmin>185</xmin>
        <ymin>282</ymin>
        <xmax>403</xmax>
        <ymax>480</ymax>
    </bndbox>
</object>
<object>
    <name>WBC</name>
    <pose>Unspecified</pose>
    <truncated>1</truncated>
    <difficult>0</difficult>
  - <bndbox>
        <xmin>1</xmin>
        <ymin>125</ymin>
        <xmax>216</xmax>
        <ymax>377</ymax>
    </bndbox>
```



```
WBC  :  99.48017001152039  :  [11, 125, 212, 372]
WBC  :  99.29207563400269  :  [183, 277, 383, 470]
```

Aquí se detectaron dos leucocitos, con resultados satisfactorios según se puede constatar con el archivo xml

Se recorta la imagen (derecha) del leucocito de coordenadas

```
WBC  :  99.48017001152039  :  [11, 125, 212, 372]
```

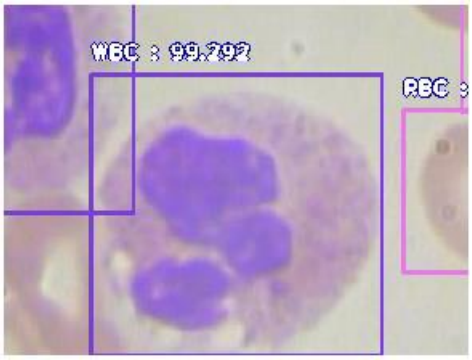
Y el clasificador indica que es:

```
[[0. 0. 0. 1.]]
[3]
neutrophil
```

El resultado es correcto. Tiene un núcleo multilobulado.

Con las coordenadas del otro leucocito:

```
WBC  :  99.29207563400269  :  [183, 277, 383, 470]
```



Y según el Clasificador:

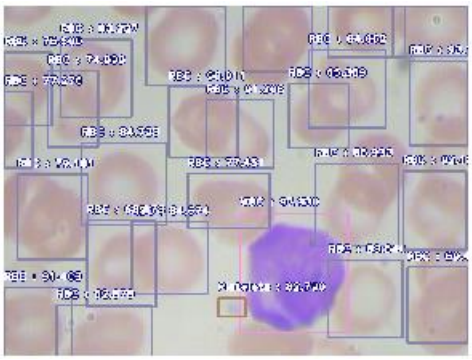
```
[[0. 0. 0. 1.]]
[3]
neutrophil
```

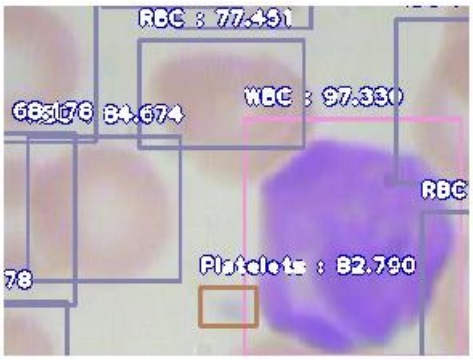
Resultado correcto, tiene un núcleo multilobulado.

Imagen 347

```
WBC  :  97.32997417449951  :  [325, 284, 474, 447]
RBC  :  94.78535652160645  :  [446, 0, 553, 70]
RBC  :  88.63220810890198  :  [536, 0, 638, 69]
RBC  :  79.96850609779358  :  [196, 0, 305, 108]
RBC  :  74.34201836585999  :  [330, 0, 424, 90]
RBC  :  83.2269549369812  :  [78, 46, 176, 152]
RBC  :  84.06169414520264  :  [420, 60, 526, 167]
RBC  :  90.91089963912964  :  [227, 111, 322, 207]
RBC  :  91.2084698677063  :  [279, 129, 371, 222]
RBC  :  84.72316265106201  :  [107, 188, 223, 300]
RBC  :  77.45109796524048  :  [253, 230, 366, 304]
RBC  :  85.93348860740662  :  [429, 216, 543, 328]
RBC  :  79.18115854263306  :  [18, 231, 115, 347]
RBC  :  84.67441201210022  :  [176, 296, 281, 395]
RBC  :  68.17794442176819  :  [114, 294, 209, 412]
RBC  :  89.54390287399292  :  [447, 348, 549, 454]
RBC  :  91.46308302879333  :  [0, 386, 94, 482]
RBC  :  52.87793278694153  :  [72, 410, 203, 482]
RBC  :  76.50049924850464  :  [1, 63, 66, 164]
RBC  :  93.17373037338257  :  [559, 75, 638, 191]
RBC  :  74.99861717224121  :  [60, 89, 130, 193]
RBC  :  65.36924839019775  :  [392, 106, 475, 195]
RBC  :  77.27620601654053  :  [1, 117, 39, 222]
RBC  :  92.1862244606018  :  [550, 226, 637, 334]
Platelets  :  82.79039859771729  :  [295, 400, 333, 426]
RBC  :  69.47208642959595  :  [556, 358, 639, 462]
```

```
<object>
    <name>WBC</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
  - <bndbox>
        <xmin>319</xmin>
        <ymin>297</ymin>
        <xmax>476</xmax>
        <ymax>453</ymax>
    </bndbox>
</object>
```






```
WBC  :  97.32997417449951  :  [325, 284, 474, 447]
```

Resultado satisfactorio.

El Clasificador da:

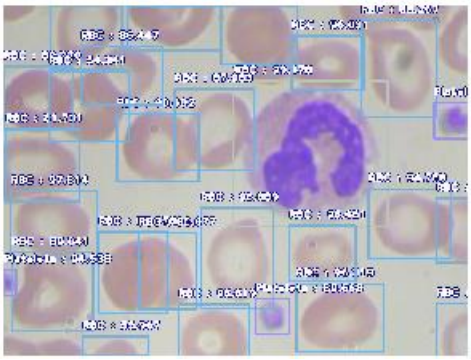
```
[[0. 0. 1. 0.]]
[2]
monocyte
```

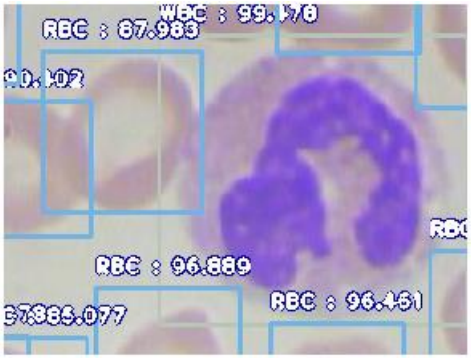
Es correcto. El núcleo no es tan fuerte para ser linfocito.

Imagen 0369

```
RBC  :  81.73081278800964  :  [3, 169, 105, 268]
RBC  :  90.4020607471466  :  [156, 148, 269, 241]
WBC  :  99.47817921638489  :  [316, 103, 529, 302]
RBC  :  97.8139340877533  :  [10, 254, 129, 347]
RBC  :  94.75826621055603  :  [503, 251, 617, 346]
RBC  :  89.04131054878235  :  [11, 338, 123, 446]
RBC  :  96.31944298744202  :  [403, 381, 523, 474]
RBC  :  56.895267963409424  :  [170, 1, 300, 61]
RBC  :  91.68395400047302  :  [64, 0, 163, 87]
RBC  :  70.51514387130737  :  [300, 0, 405, 80]
RBC  :  96.03124856948853  :  [397, 41, 496, 115]
RBC  :  88.26685547828674  :  [493, 20, 597, 153]
RBC  :  86.44747138023376  :  [107, 56, 219, 136]
RBC  :  89.21979665756226  :  [0, 83, 95, 170]
RBC  :  54.34137582778931  :  [62, 90, 173, 186]
RBC  :  87.98269629478455  :  [236, 114, 345, 225]
RBC  :  96.88938856124878  :  [272, 277, 372, 414]
RBC  :  96.46100401878357  :  [393, 301, 485, 377]
RBC  :  89.47803974151611  :  [131, 310, 226, 421]
RBC  :  98.81935715675354  :  [241, 409, 338, 482]
Platelets  :  89.94930982589722  :  [593, 133, 640, 182]
RBC  :  65.65039157867432  :  [596, 264, 642, 352]
RBC  :  85.07692813873291  :  [186, 311, 266, 416]
Platelets  :  94.53813433647156  :  [0, 362, 17, 397]
Platelets  :  89.66754078865051  :  [347, 402, 393, 450]
RBC  :  85.10964512825012  :  [597, 408, 640, 482]
RBC  :  55.96718192100525  :  [109, 453, 199, 479]
```

```
<object>
    <name>WBC</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
  - <bndbox>
        <xmin>324</xmin>
        <ymin>107</ymin>
        <xmax>522</xmax>
        <ymax>303</ymax>
    </bndbox>
</object>
```






```
WBC  :  99.47817921638489  :  [316, 103, 529, 302]
```

Resultado satisfactorio

El clasificador:

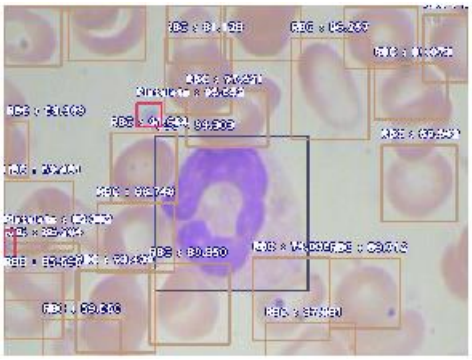
```
[[0. 0. 0. 1.]]
[3]
neutrophil
```

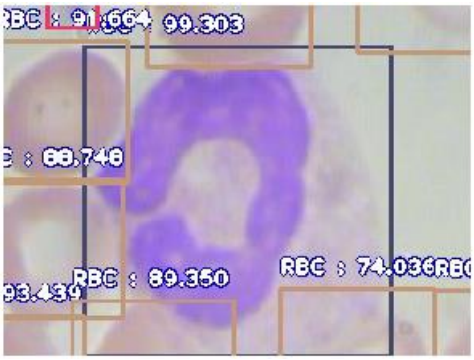
El resultado es correcto.

Imagen 0349

```
WBC  :  99.30282831192017  :  [208, 179, 419, 391]
RBC  :  85.46646237373352  :  [89, 2, 195, 75]
RBC  :  94.13586258888245  :  [304, 0, 409, 76]
RBC  :  90.26403427124023  :  [469, 2, 570, 86]
RBC  :  81.12847208976746  :  [222, 45, 313, 149]
RBC  :  96.25740051269531  :  [396, 45, 503, 183]
RBC  :  89.07349705696106  :  [510, 79, 611, 156]
RBC  :  97.62083292007446  :  [251, 116, 364, 193]
RBC  :  91.66362881660461  :  [147, 175, 238, 272]
RBC  :  89.54222798347473  :  [520, 191, 624, 296]
RBC  :  69.4812536239624  :  [0, 240, 95, 336]
RBC  :  88.74847888946533  :  [128, 270, 235, 365]
RBC  :  86.20449304580688  :  [1, 325, 103, 404]
RBC  :  74.03631210327148  :  [343, 345, 449, 458]
RBC  :  89.71206545829773  :  [449, 346, 546, 442]
RBC  :  93.43860745429993  :  [99, 363, 208, 475]
RBC  :  89.3503189086914  :  [200, 353, 311, 464]
RBC  :  59.56611633300781  :  [54, 430, 161, 483]
RBC  :  87.08096742630005  :  [360, 435, 484, 481]
RBC  :  96.46456837654114  :  [0, 2, 79, 83]
RBC  :  71.34070992469788  :  [225, 0, 298, 79]
RBC  :  91.25874638557434  :  [578, 13, 639, 105]
Platelets  :  95.65213918685913  :  [183, 134, 217, 166]
RBC  :  59.308767318725586  :  [4, 159, 34, 235]
Platelets  :  82.3815643787384  :  [0, 308, 15, 346]
RBC  :  85.69605946540833  :  [0, 366, 81, 456]
```

```
<object>
    <name>WBC</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
  - <bndbox>
        <xmin>202</xmin>
        <ymin>181</ymin>
        <xmax>419</xmax>
        <ymax>392</ymax>
    </bndbox>
</object>
```






```
WBC  :  99.30282831192017  :  [208, 179, 419, 391]
```

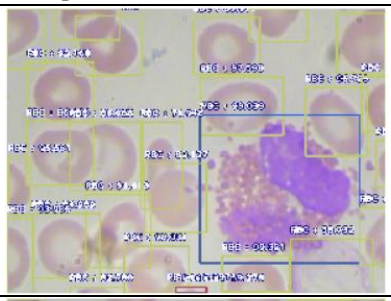
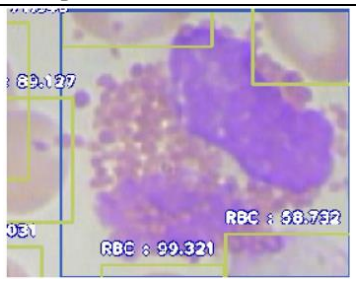
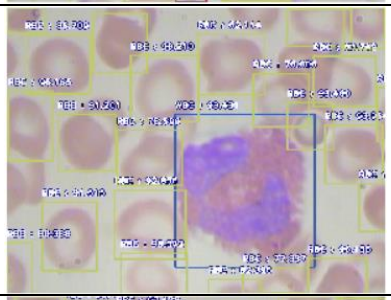
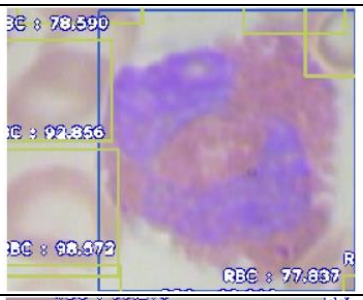
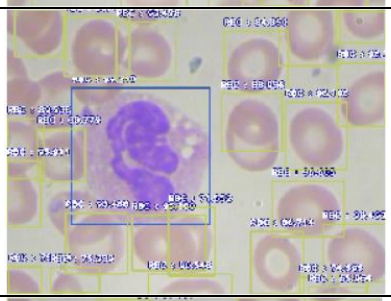
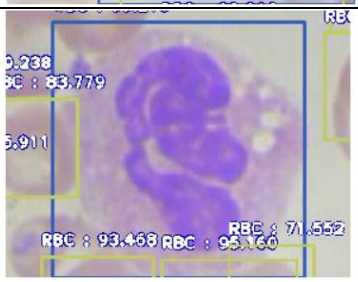
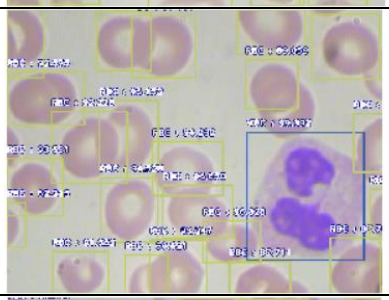
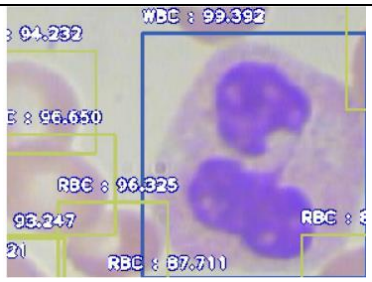
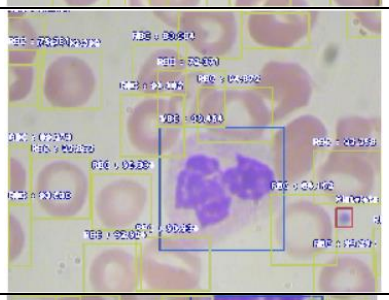
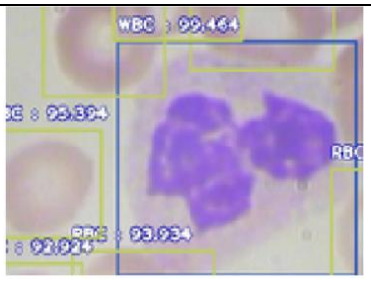
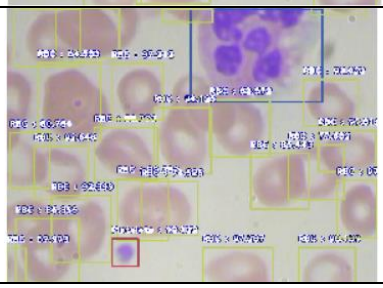
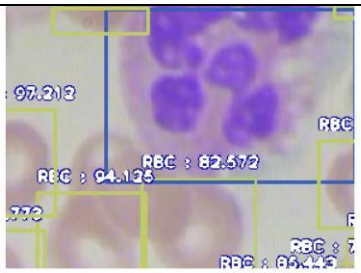
Resultado satisfactorio

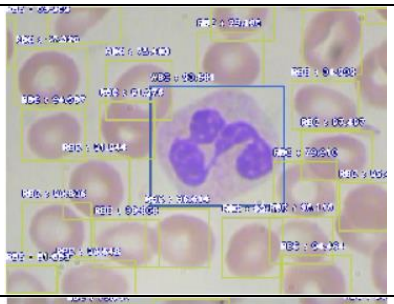
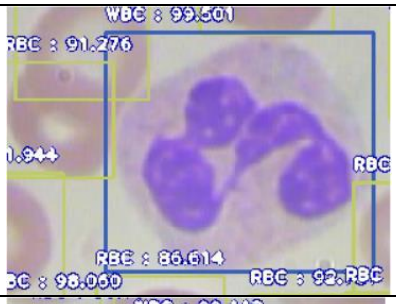
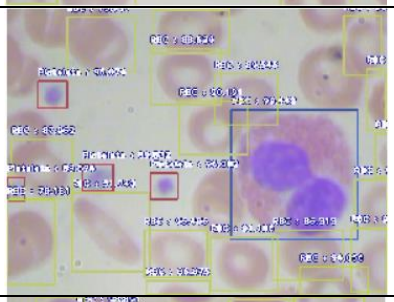
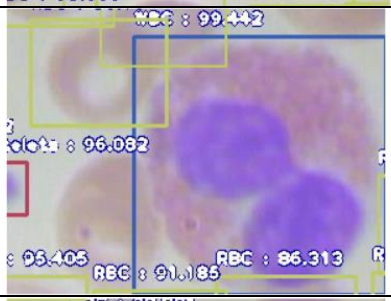
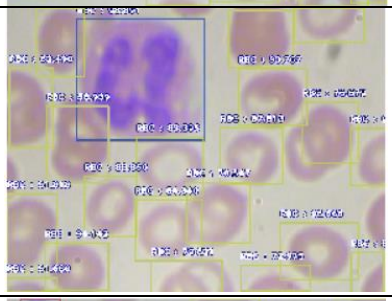
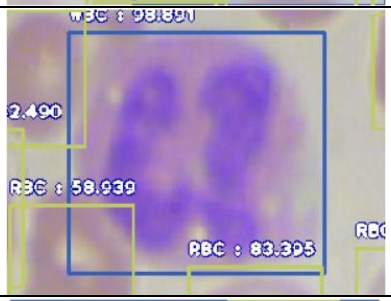
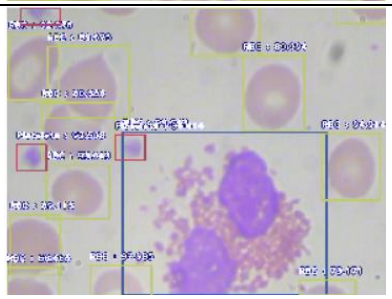
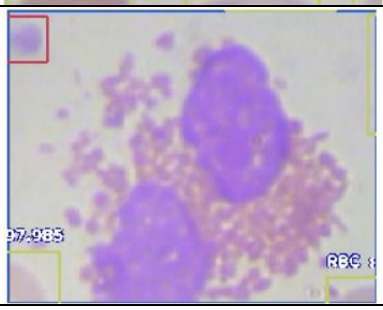
Clasificador:

```
[[0. 0. 0. 1.]]
[3]
neutrophil
```

El resultado es correcto.

Para las siguientes imágenes, se realiza una tabla de las imágenes probadas y los resultados:

Imagen	Imagen Detector	Imagen leucocito	Tipo
0352			Neutrophil. (Incorrecto, eosinófilo)
0353			eosinophil **(Correcto)**
0354			Neutrophil (Correcto)
0357			Neutrophil. (Correcto)
0360			Neutrophil (Correcto)
0364			Neutrophil. (Correcto)

0370			Neutrophil (Correcto)
0410			Monocyte (Correcto)
0400			Neutrophil (Correcto)
0383			Neutrophil (Incorrecto, Eosinófilo)