UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

FCEN
FACULTAD DE CIENCIAS EXACTAS Y NATURALES

Facultad de Ciencias Exactas y Naturales. UNCUYO

Estudio de técnicas de Aprendizaje Automático en el análisis de imágenes para pronóstico de cosecha

Seminario de Investigación y/o Desarrollo Tecnológico
para optar al título de grado de la
Licenciatura en Ciencias Básicas con orientación en Matemática

Estudiante: Tatiana Sofía Parlanti

Director: Dr. Emmanuel Nicolás Millán

Codirector: Dr. Alejandro Martín Lobos

Mendoza, Argentina
Abril 2021

A Mamá María

Agradecimientos

A mi familia, gracias por el constante soporte y amor de siempre. Por respetar mis decisiones y alentarme cuando los resultados no eran los proyectados. Al Agus por su apoyo incondicional en esta última etapa, por la *casi* infinita paciencia, la empatía y el aliento desmesurado, y a mi *otra* familia, por su cariño y generosidad al abrirme las puertas de su hogar.

A mi director Emmanuel, gracias por este tiempo compartido, por aceptar dirigirme aún sabiendo que yo "no era del palo", por enseñarme con paciencia y responder mis muchas preguntas. Gracias porque en este tiempo de pandemia, cuando ya no fue posible trabajar en la oficina y se dificultó el acceso remoto al cluster, no faltaron las palabras motivadoras frente a las problemáticas que surgían. Gracias por introducirme al mundo apasionante del Aprendizaje Automático, y acompañarme en el proceso de buscar nuevos temas de investigación.

A mi codirector Alejandro, gracias por sumarte al proyecto y por la buena predisposición de siempre.

A los jurados Martín y Sebastián, gracias por darse el tiempo de leer este trabajo, por sus comentarios y observaciones que ayudaron a mejorarlo.

A mis amigos, de la facu y de la vida, por compartir alegrías y tristezas y hacer más ameno el camino. En especial, a Anita, Ro y Vicky, grandes amigas y compañeras de carrera, su apoyo en este último tiempo ha sido esencial para mí.

A los Ingenieros Agrónomos Marcos Montoya y Julieta Dalmasso del INTA. A la familia Scattareggia y Finca Coletto por darnos acceso a sus fincas para fotografiar los espalderos. Al CONICET y la SIIP-UNCUYO por el financiamiento al proyecto donde está enmarcado este Seminario de Investigación y/o Desarrollo Tecnológico. Al CIN, porque este trabajo se realizó en el marco de una Beca Estímulo a las Vocaciones Científicas.

A todos los que ayudaron directa o indirectamente a que este trabajo se completara.

A mi Lela, con cariño,

es sólo un escaloncito más.

Resumen

Una de las necesidades de los productores de la región es contar con un proceso adecuado de pronóstico de cosecha de uva. Tradicionalmente, la metodología más precisa y rápida para llevar a cabo este pronóstico está dada por el recuento de racimos por planta y la estimación de su peso. La extrapolación del resultado unitario a todas las plantas teniendo en cuenta datos históricos determina el valor de producción proyectado. Este proceso implica la utilización de recursos humanos, materiales y tiempo que, por su propia escasez, limitan el proceso. En este trabajo buscamos incorporar tecnología en el proceso de estimación de cosecha de uva, mediante la captura de imágenes, su pre-procesamiento y etiquetado y su posterior análisis usando algoritmos computacionales de Aprendizaje Automático (AA). Para ello estudiaremos sobre AA y algoritmos de aprendizaje supervisado, en particular el uso de Redes Neuronales Convolucionales para abordar la tarea de detectar objetos (racimos de uva), en las imágenes capturadas. Pondremos a prueba la red YOLOv3 (por sus siglas en inglés *You Only Look Once*, que significa "sólo miras una vez") con diferentes conjuntos de fotografías, aplicaremos técnicas de aumento de datos, usaremos diferentes valores de hiperparámetros y arquitecturas de hardware, y reportaremos medidas de rendimiento como mAP, exhaustividad y precisión. Veremos por ejemplo que el aplicar distintas técnicas de aumento de datos lleva a un incremento de la exhaustividad de entre el 1 y el 13 %, mientras que el incremento al usar diferentes hiperparámetros puede ser de entre un 5 y un 8 %. Además veremos que logramos detectar correctamente un 96,1 % de racimos con una precisión del 72,16 % para el entrenamiento con mayor número de fotografías.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

El presente escrito representa la culminación del Seminario de Investigación y/o Desarrollo Tecnológico para la obtención del título de grado de Licenciatura en Ciencias Básicas con orientación en Matemática de la FCEN-UNCUYO. En este primer capítulo introducimos los antecedentes y motivación del problema a resolver, así como los objetivos e hipótesis y la organización del trabajo en sí.

1.1. Motivación

El Aprendizaje Automático (AA o ML por sus siglas en inglés *Machine Learning*) es el campo de estudio que le permite a las computadoras tener la habilidad de aprender sin haber sido programadas explícitamente para ello [76]. Se suele relacionar el AA con la detección automática de patrones o información útil en datos. El uso del AA para el análisis de datos se presenta como un tema ampliamente utilizado en diversas áreas, desde las sociales, hasta la salud y la ingeniería [34, 23]. En particular, se utiliza en el reconocimiento automático de objetos, que tiene aplicaciones directas en los procesos agrícolas.

Una necesidad local es contar con un proceso adecuado de pronóstico de cosecha de uva. Tradicionalmente, la metodología más precisa y rápida para llevar a cabo este pronóstico está dada por el recuento de racimos por planta y la estimación de su peso. La extrapolación del resultado unitario a todas las plantas teniendo en cuenta datos históricos determina el valor de producción proyectado. También se utiliza la estimación visual directa pero es subjetiva. Automatizar el análisis de imágenes de vid para asistir en la estimación del pronóstico de cosecha de la misma se ve como un problema que puede ser abordado desde el AA.

La estimación de la producción agrícola en general y de la vid en particular, es necesaria para la planificación que deben llevar a cabo tanto los actores del sector público como los del sector privado. La implicación es muy amplia, desde una perspectiva social, es necesario recopilar información por razones de seguridad alimentaria y sustentabilidad ambiental; y desde una perspectiva de gestión de cosecha, la información permite desarrollar una organización confiable, anticipada y oportuna de rendimiento de los cultivos. Además esto influye en aspectos tales como transporte, almacenamiento, importación/exportación y comercialización, ahorrando costos, aportando mayor eficiencia y por lo tanto mejorando rendimientos.

1.2. Objetivos

En este Seminario de Investigación y/o Desarrollo Tecnológico planteamos analizar, tanto desde un punto de vista matemático como del computacional, imágenes de vid con algoritmos de AA para detectar el fruto (granos de uva) con el objetivo de cuantificarlos y asistir con el pronóstico de cosecha. Los objetivos específicos incluyen:

1. Estudiar algoritmos supervisados para la detección de granos de uva en imágenes.
2. Evaluar diversas métricas de desempeño de los algoritmos para compararlos y así poder evaluar sus fortalezas y limitaciones.
3. Estudiar y analizar la matemática subyacente de los algoritmos utilizados con el objetivo de ajustar los hiperparámetros de los modelos para mejorar su desempeño a través de las métricas estudiadas.

4. Aplicar lo desarrollado en la tesis sobre imágenes de campo.

1.3. Hipótesis y resultados esperados

Detallamos a continuación las hipótesis formuladas a partir de los objetivos planteados:

1. Es posible utilizar algoritmos de AA para identificar de forma confiable en imágenes los frutos de la planta de la vid.

2. Es posible cuantificar (con cierto margen de error) el pronóstico de cosecha utilizando las imágenes capturadas, a través de la identificación de cada grano de uva.

3. Es viable mejorar los resultados obtenidos y/o el tiempo de cómputo necesario para obtenerlos, mediante la optimización del espacio de hiperparámetros de los algoritmos de Aprendizaje Automático.

Además esperamos obtener los siguientes resultados:

1. Reporte del estado del arte de los algoritmos de Aprendizaje Automático para la detección de objetos en imágenes.

2. Implementación de algoritmos supervisados como Redes Neuronales y sus variantes en infraestructura de Computación de Alto Desempeño (HPC).

3. Análisis de imágenes de plantas de vid para la detección del fruto (uva) que permita cuantificar y pronosticar la producción del mismo sobre las imágenes tomadas.

4. Análisis de rendimiento de los algoritmos de Aprendizaje Automático utilizados en varias arquitecturas HPC, utilizando diversos parámetros como tiempo de ejecución y métricas de rendimiento como eficiencia y precisión.

1.4. Antecedentes

El problema de detección de objetos en imágenes consiste en determinar dónde se encuentran los objetos en una imagen dada (regresión) y a qué categoría pertenece cada uno (clasificación). Por lo que el proceso que llevan a cabo los algoritmos de detección de objetos, en general, puede dividirse en tres etapas: selección de una región de interés, extracción de características y clasificación. Si bien para las tareas de clasificación se pueden usar Máquinas de Soporte Vectorial (SVM por sus siglas en inglés *Support Vector Machines*) o Bosques Aleatorios (RF por sus siglas en inglés *Random Forest*), lo cierto es que con la aparición de las Redes Neuronales Convolucionales (CNN por sus siglas en inglés *Convolutional Neural Networks*) [11] y las Redes Neuronales Profundas (DNN por sus siglas en inglés *Deep Neural Networks*) [34], que tienen arquitecturas con capacidad para aprender

características más complejas, las tareas de detección de objetos en imágenes se han desarrollado principalmente para este tipo de Redes Neuronales [87]. En particular desde que se propuso el detector R-CNN [29] que combina la propuesta de regiones de interés con CNN, se han sugerido una gran cantidad de versiones mejoradas, entre las que se incluyen Fast R-CNN [32], que optimiza conjuntamente las tareas de clasificación y regresión del rectángulo delimitador, y Faster R-CNN [42], que usa una subred adicional para proponer las regiones de interés, mientras que la red YOLO [41] también usa CNN y logra la detección de objetos al dividir la imagen en una cuadrícula y predecir rectángulos delimitadores en cada celda.

Existen diversas aplicaciones del uso de estos algoritmos para detectar objetos en imágenes, desde el reconocimiento facial [12, 44], el reconocimiento de peatones [22] y la conducción autónoma de vehículos [31], hasta la detección en problemas de salud [10, 62]. Un área de aplicación es en la agricultura, en particular para la detección de frutos. Por ejemplo en [47] plantean la detección de frutas en huertos, incluyendo mangos, almendras y manzanas usando Faster R-CNN, mientras que en [78] y en [85] usan YOLOv3 para la detección de frutos de mango en imágenes de copas de árboles, y la identificación de manzanas en diferentes momentos de crecimiento, respectivamente.

Para el caso de la detección del fruto de la vid, en los últimos años se han visto avances en la estimación del pronóstico de cosecha, utilizando técnicas de AA. Por ejemplo, Font et al. presentaron en [28] un método automático basado en la detección de los picos de reflexión especular de la superficie esférica de las uvas, para el recuento de éstas a partir de imágenes de alta resolución de viñedos tomadas con iluminación artificial por la noche, lo que significa que los operadores tienen que trabajar en una ventana de tiempo inusual. Por otro lado, Abdelghafour et al. determinaron la afiliación de un píxel a un racimo de uva basándose en características colorimétricas y de textura, utilizando un clasificador supervisado SVM en [45] para la detección y la medición de los racimos de uva entre la época de floración y los primeros estadios de fructificación. Aquino et al. presentaron en [46] un algoritmo basado en la morfología matemática y la clasificación de píxeles para el recuento de granos de uva tomando las fotografías colocando una cartulina oscura detrás del racimo, lo cual es difícil de implementar para un operador. Luego, estimaron en [54] el pronóstico de cosecha basado en imágenes RGB adquiridas "sobre la marcha" desde un vehículo todoterreno utilizando iluminación artificial en la noche, que al igual que con [28] implica un horario de trabajo inusual para los operadores. También presentaron vitisBerry [55], una aplicación para *smartphones* que permite evaluar en el viñedo, mediante visión artificial, el número de granos de uva en los racimos en los estados fenológicos entre el cuajado (caída de los capuchones florales) y el cierre del racimo. Por su parte, Coviello et al. teniendo en cuenta los limitantes antes mencionados, presentaron en [56, 75] una estimación del pronóstico de cosecha de uva usando una adaptación de algoritmos de conteo de multitudes, mientras que Zabawa et al. detectaron granos de uva en imágenes usando

CNN en [86].

1.5. Implementación

Los pasos relevantes para automatizar el proceso de pronóstico de cosecha son, en primer lugar, la adquisición de imágenes, luego la identificación automática de racimos y, por último, el análisis de éstos para realizar un pronóstico de cosecha. En este trabajo nos centramos en las primeras etapas de este proceso: adquisición de imágenes, pre-procesamiento y etiquetado de las mismas, entrenamiento y detección con Redes Neuronales y post-análisis de los resultados.

Para ello estudiamos sobre AA y algoritmos de aprendizaje supervisado, en particular el uso de Redes Neuronales Convolucionales (CNN) para abordar tareas de detección de objetos en imágenes. Si bien el funcionamiento de muchos de los algoritmos de AA se fundamenta en conceptos y propiedades de Álgebra Lineal y Cálculo Multivariable, no deja de ser un área de las Ciencias de la Computación, donde en muchos casos no se requiere saber en profundidad de estos conceptos Matemáticos para su implementación en algún problema. Sin embargo, uno de nuestros objetivos fue estudiar el trasfondo de dichos algoritmos para entender, desde un punto de vista formal, cómo se logra el aprendizaje de las computadoras. Estos conceptos son los que hemos querido plasmar en el Capítulo 2, que hacen a la justificación del funcionamiento de la red YOLOv3, que usamos para detectar racimos de uva en fotografías tomadas de espalderos.

Por otro lado, además de este estudio, fue necesaria la adquisición de imágenes de campo, para lo cual nos trasladamos a distintas fincas para tomar fotografías con teléfonos celulares y capturar videos con drones y celulares. Para realizar el pre-procesamiento de las imágenes aprendí a usar herramientas de `Linux` para manipulación masiva de archivos (`Bash`), y de `Python` para llevar a cabo el etiquetado a mano de cada fotografía. Una limitación común de las Redes Neuronales (NN) es que suelen requerir de cientos a miles de imágenes de entrenamiento para la correcta detección de objetos. Como en nuestro caso no pudimos recolectar tal cantidad de imágenes, aplicamos una serie de transformaciones para aumentar el número de datos. Luego entrenamos NN usando la estructura de YOLOv3 e hicimos dos tipos de pruebas principales: por un lado entrenamos modelos de detección con 100 y 500 fotografías, y por otro entrenamos modelos con las imágenes transformadas del primer conjunto de 100 fotos. Finalmente comparamos el desempeño de los distintos modelos haciendo un post-análisis de los resultados obtenidos, para calcular medidas de rendimiento como exhaustividad y precisión.

1.6. Organización

El presente trabajo está organizado como sigue: el Capítulo 2 nuclea las nociones principales de AA, NN y CNN necesarias para comprender el funcionamiento de YOLOv3,

que es la red que usamos para entrenar los distintos modelos, así como la explicación de distintos tipos de tareas que se resuelven con estos algoritmos, en particular la detección de objetos en imágenes. A su vez, en el Capítulo 3 presentamos la obtención de imágenes, así como el proceso de etiquetado y aplicación de transformaciones a las mismas, junto con las especificaciones técnicas y la descripción del hardware y software utilizado. Finalmente, exponemos los principales resultados y discusiones en el Capítulo 4, así como las conclusiones en el Capítulo 5.

Los principales términos o conceptos están enfatizados con MAYÚSCULAS PEQUEÑAS. Además, como la bibliografía de estos temas se encuentra escrita principalmente en inglés, junto con cada término aclaramos entre paréntesis el nombre correspondiente en esa lengua usando *letras itálicas*, y usamos también principalmente las siglas en inglés.

Capítulo 2

Aprendizaje Automático

Este capítulo tiene el propósito de introducir las nociones básicas del Aprendizaje Automático, y en particular de Redes Neuronales, y no asume que el lector sepa de estos temas de antemano. El objetivo es explicar todo lo necesario para concluir el capítulo centrándonos en entender el funcionamiento de la Red Neuronal YOLO, la cual usamos en este Seminario de Investigación y/o Desarrollo Tecnológico para detectar racimos de uva en imágenes.

Comenzaremos definiendo el Aprendizaje Automático y nombrando los tipos de aprendizaje que existen (Sección 2.1), luego explicaremos las principales tareas del aprendizaje supervisado, que son la regresión y clasificación y representan la base de los problemas de detección de objetos en imágenes (Sección 2.2). Posteriormente veremos la estructura y base del aprendizaje de las Redes Neuronales (Sección 2.3), así como de las Redes Neuronales Convolucionales (Sección 2.4). Finalmente explicaremos en qué consiste la detección de objetos y cómo YOLO encara y resuelve este tipo de tareas.

Cada concepto importante en español tiene aclarado entre paréntesis su nombre en inglés, ya que la bibliografía de la mayoría de estos temas se encuentra escrita en esta lengua.

2.1. ¿Qué es el Aprendizaje Automático?

El Aprendizaje Automático (AA o ML por sus siglas en inglés *Machine Learning*), es la ciencia que estudia cómo lograr que las computadoras aprendan a partir de los datos, es decir se trata de una situación en la que no se cuenta con una solución analítica pero se utilizan datos para construir una solución empírica. Arthur Samuel [8] definió en 1959 el AA como el campo de estudio que le permite a las computadoras tener la habilidad de aprender sin haber sido programadas explícitamente para ello. Más tarde, en 2006 Tom Mitchell [16] lo definió como la ciencia de construir sistemas computacionales que mejoran automáticamente con la experiencia y estudian cuáles son las leyes fundamentales que gobiernan todos los procesos de aprendizaje.

Las técnicas tradicionales de programación implican estudiar un problema a resolver, escribir reglas, evaluar y analizar errores para ajustar dichas reglas. Cuando nos enfrentamos a problemas más complejos, como clasificación de objetos o predicción de valores, un programa tradicional necesitaría una muy extensa lista de comandos, probablemente difícil de actualizar frente a nuevos cambios en los datos. En cambio, un programa basado en técnicas de AA aprende a reconocer automáticamente cuáles son los predictores buenos, es decir las variables que explican o predicen el valor observado, o a detectar patrones de frecuencia inusual en los datos. Este tipo de programa es mucho más corto, fácil de actualizar y probablemente más preciso [76].

Además, el AA se utiliza en la resolución de problemas que no tienen un algoritmo conocido, como es el caso del reconocimiento de voz; y problemas que involucran grandes cantidades de datos. A su vez, los algoritmos de AA pueden ser inspeccionados para ver qué han aprendido, es decir, ver cuáles son los predictores que considera "buenos". Muchas veces, a partir de esto se descubren patrones que no se habían tenido en cuenta inicialmente, que pueden ayudar a comprender mejor el problema.

Los datos que se utilizan para que las computadoras aprendan son la entrada (*input*) que necesita el sistema para aprender, y se conoce como conjunto de entrenamiento (aunque no es un conjunto propiamente dicho, ya que puede contener elementos repetidos y algunos algoritmos pueden tener en cuenta el orden de los elementos del conjunto [30]). A partir de estos, el algoritmo determina una función de predicción $g : \mathcal{X} \to \mathcal{Y}$, donde $\mathcal{X}$ es el conjunto de entrada (entrenamiento) e $\mathcal{Y}$ el de salida, y mide el error. Decimos que la computadora aprende cuando busca minimizar el error cometido. Más precisamente, de acuerdo a la definición de Mitchell [16], decimos que una máquina aprende con respecto a una tarea específica T, una métrica de rendimiento P y un tipo de experiencia E, si mejora sustancialmente su rendimiento P para realizar la tarea T a partir de los ejemplos E.

A continuación daremos la diferencia entre parámetros e hiperparámetros, ya que a lo largo de este trabajo hablaremos de estos dos conceptos, y es importante distinguirlos bien.

2.1.1. Parámetros e hiperparámetros

En el AA la función $g : \mathcal{X} \to \mathcal{Y}$ se denomina MODELO. Un PARÁMETRO del modelo es una variable de configuración interna cuyo valor puede ser estimado a partir de los datos. Se puede pensar en el modelo como una hipótesis y en los parámetros como la adaptación de la hipótesis a un conjunto específico de datos [76, 73].

A menudo los parámetros del modelo se estiman utilizando un algoritmo de optimización, que es un tipo de "búsqueda" eficiente a través de posibles valores que pueden tomar. El modelo requiere de dichos parámetros para hacer predicciones, y su principal característica es que son aprendidos o estimados a partir de los datos, por lo que no son establecidos manualmente por una persona, y son almacenados como parte del modelo aprendido.

Por otro lado, un hiperparámetro es una configuración externa al modelo y cuyo valor no puede ser estimado a partir de los datos. No se puede saber el mejor valor de un hiperparámetro en un problema dado, pero se utilizan reglas generales, como copiar los valores utilizados en otros problemas, o buscar el mejor valor por ensayo y error [76, 73].

Los hiperparámetros se utilizan en procesos para ayudar a estimar los parámetros del modelo, y en general son fijados por una persona. En nuestro caso, especificaremos los hiperparámetros elegidos para entrenar modelos en el Capítulo 3.

Existen diferentes tipos de algoritmos de AA, que se pueden clasificar en grandes grupos. Una de las clasificaciones principales diferencia los algoritmos supervisados de los no supervisados, mientras que otra se basa en el momento en que los parámetros son actualizados.

2.1.2. Tipos de aprendizaje automático

Teniendo en cuenta la cantidad y tipo de supervisión que los algoritmos reciben durante el entrenamiento, se diferencian algoritmos de aprendizaje supervisado, aprendizaje no supervisados y de aprendizaje con refuerzo [76, 49]. Por otro lado, otro criterio utilizado para clasificar algoritmos de AA es en qué momento pueden actualizar los parámetros del modelo, entre los cuales se diferencian los algoritmos de aprendizaje *online* de los de aprendizaje *offline* [76, 18].

Aprendizaje supervisado

El conjunto de entrenamiento en el APRENDIZAJE SUPERVISADO contiene ejemplos explícitos de cuáles son las salidas correctas para cada dato de entrada, es decir, está formado por pares $(entrada, salida\,correcta)$. Dichas salidas deseadas son llamadas ETIQUETAS. El objetivo de estos algoritmos es determinar correctamente cuál es la etiqueta correspondiente para ejemplos desconocidos (es decir, que no forman parte del conjunto de entrenamiento). Para esto determina una función g que se ajusta a los datos y un "crítico" compara las salidas obtenidas con las etiquetas para calcular el error cometido, buscando luego la

minimización de éste. El esquema a la izquierda de la Figura 2.1 representa este tipo de aprendizaje.

Dado que para este trabajo nos enfocaremos en técnicas de aprendizaje supervisado, daremos una descripción más detallada de éste en la Sección 2.1.3.

Aprendizaje no supervisado

El APRENDIZAJE NO SUPERVISADO se ocupa de encontrar patrones y estructuras en los datos no etiquetados. En estos casos, como no se dispone del resultado deseado, el sistema trata de aprender "sin un maestro" segmentando los datos en "clases" adecuadas, en el sentido en que divide a los datos en grupos según alguna/s característica/s que determina el algoritmo (no se trata de las clases propias de una tarea de clasificación). El esquema del medio de la Figura 2.1 representa al aprendizaje no supervisado. A continuación mencionamos algunos de los ejemplos más comunes en este tipo de aprendizaje.

- Conglomerado (*clustering*): estos algoritmos detectan grupos de datos similares, sin intervención del usuario. Algunos ejemplos son *K-Means, DBSCAN, Hierarchical Cluster Analysis (HCA).*
- Detección de anomalías o novedades: el conjunto de entrenamiento en el caso de detección de anomalías consiste de ejemplos "normales", por lo que el sistema aprende a reconocerlos de manera tal que cuando se le presenta un nuevo ejemplo puede determinar si se trata de algo usual o de una anomalía (por ejemplo, detectar transacciones inusuales de tarjetas de crédito). La detección de novedades es similar a la anterior, con la diferencia de que su conjunto de entrenamiento puede contener una pequeña cantidad de ejemplos atípicos. Ejemplos de estos algoritmos son *One-class SVM, Isolation Forest.*
- Visualización y reducción de dimensión: el objetivo es obtener una representación en el plano o en el espacio de los datos de entrada. Estos algoritmos tratan de preservar tanta estructura como sea posible (por ejemplo, manteniendo los distintos *clusters* separados para una correcta visualización), para que podamos comprender cómo están organizados los datos (y notar patrones no detectados "a simple vista"). Por otro lado, el objetivo de la reducción de dimensión es simplificar los datos sin perder demasiada información, por ejemplo combinando características relacionadas. Algunos ejemplos de estos algoritmos son *Principal Component Analysis (PCA), Kernel PCA, Locally-Linear Embedding (LLE), t-distributed Stochastic Neighbor Embedding (t-SNE).*

Aprendizaje con refuerzo

En el APRENDIZAJE CON REFUERZO se agrega una recompensa o castigo para mejorar el proceso, emulando algunas formas de aprendizaje natural. El sistema aprende interac-

tuando con un ambiente y cambiando su comportamiento para maximizar su recompensa. En este caso el conjunto de entrenamiento contiene estados en un ambiente y las acciones posibles para un estado determinado, es decir, no contiene la salida correcta para cada entrada, sino que tiene algunas posibles salidas junto con una medida de qué tan buenas son. Durante el proceso de aprendizaje el algoritmo explora aleatoriamente los pares estado-acción dentro de algún entorno y luego, según lo aprendido, utiliza las recompensas del par estado-acción para elegir la mejor acción que, dado un estado, lleve a algún estado objetivo final.

Un ejemplo de uso es aprender cómo se juega un juego, colocando en el conjunto de entrenamiento la información de estados posibles de juego junto con las jugadas (acciones) que llevan a ganar un partido, y aquellas que llevan a perderlo. A diferencia del aprendizaje supervisado, en el que un crítico externo califica cada ejemplo, en el aprendizaje con refuerzo, este crítico sólo puede dar una calificación cuando se alcanza el estado objetivo final. El esquema a la derecha de la Figura 2.1 representa este tipo de aprendizaje.

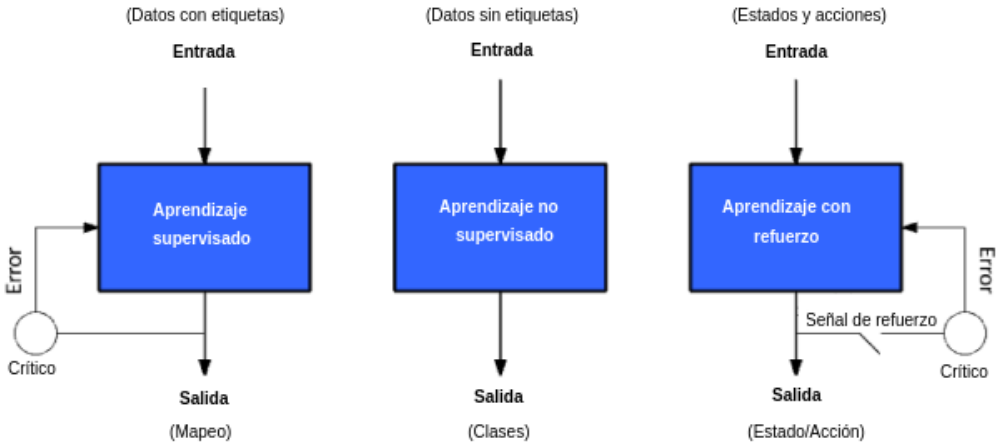


Figura 2.1. Tipos de aprendizaje automático.[1]

Aprendizaje *offline* y *online*

El aprendizaje puede ser *offline*, esto es luego de acceder a TODO el conjunto de entrenamiento, los parámetros son actualizados y se calcula el error total. Si se obtuvieran nuevos ejemplos es necesario repetir la etapa de entrenamiento con todos los datos (ya sean los nuevos como los anteriores), a fin de obtener un modelo actualizado para reemplazar al anterior. Este tipo de aprendizaje toma mucho tiempo y requiere de muchos recursos

[1] Imagen adaptada de [49].

computacionales.

Por otro lado, el aprendizaje puede ser *online*, es decir, los parámetros se actualizan luego de acceder a CADA ejemplo del conjunto de entrenamiento, o a un pequeño grupo de datos llamado MINI LOTE (*mini batch*). Cada paso de aprendizaje es rápido, por lo que el sistema puede aprender a partir de datos nuevos "sobre la marcha".

En este trabajo usaremos aprendizaje por mini lotes, así como del tipo supervisado. De este último daremos una descripción detallada a continuación.

2.1.3. Elementos del aprendizaje supervisado

Los algoritmos de AA del tipo supervisado requieren de ciertos elementos, como un conjunto de datos etiquetados, un conjunto de hipótesis y una función de costo, los cuales definimos a continuación siguiendo a [24, 80, 30].

Definición 2.1.1. Sean $N, M, n \in \mathbb{N}$. Sea $f : \mathcal{X} \to \mathcal{Y}$ una función desconocida, donde $\mathcal{X}$ es el espacio de entrada, que consta de n posibles entradas $\mathbf{x}^\mu \in \mathbb{R}^N$, e $\mathcal{Y}$ el espacio de todas las salidas posibles. Llamamos CONJUNTO DE DATOS (*dataset*) a $\mathcal{D}$, formado por todos los pares de datos entrada-salida

$$\mathcal{D} = (\mathbf{X}, \mathbf{Y}) = \{(\mathbf{x}^\mu, \mathbf{y}^\mu) \mid \mathbf{y}^\mu = f(\mathbf{x}^\mu),\ \mu = 1, 2, \ldots, n\},$$

donde $\mathbf{X} \in \mathbb{R}^{n \times N}$ es la matriz de variables independientes que tiene por filas las entradas $\mathbf{x}^\mu$, e $\mathbf{Y} \in \mathbb{R}^{n \times M}$ es la matriz de variables dependientes, cuyas filas son las correspondientes salidas $f(\mathbf{x}^\mu) \in \mathbb{R}^M$. Llamamos a f FUNCIÓN OBJETIVO.
Vamos a suponer que cada componente de las salidas $\mathbf{y}^\mu$ son valores reales, incluso en problemas de clasificación, donde se espera que la salida sean nombres como "perro" o "gato", ya que se asocia un número diferente por cada clase. En muchos problemas la salida deseada está representada por un solo valor ($M = 1$), sin embargo es posible que por cada entrada $\mathbf{x}^\mu$, la salida asociada se represente con más de uno, como es el caso de detección de objetos en imágenes, tema de interés central en este Seminario de Investigación, donde las salidas deseadas están representadas por: nombre de la clase a la cual pertenece el objeto y coordenadas del centro del rectángulo delimitador, junto con el ancho y alto del mismo, o nombre de la clase y coordenadas de los vértices superior izquierdo e inferior derecho.

Definición 2.1.2. Sea $\mathcal{X}$ el dominio de la función objetivo desconocida f y sea $\boldsymbol{\Theta}$ el espacio de parámetros. Llamamos CONJUNTO DE HIPÓTESIS a

$$\mathcal{H} = \{h_{\boldsymbol{\theta}}(\mathbf{x}) \mid \mathbf{x} \in \mathcal{X} \ \wedge\ \boldsymbol{\theta} \in \boldsymbol{\Theta}\},$$

formado por las funciones h que predicen un valor de salida a partir de un vector de variables de entrada. Estos modelos $h_{\boldsymbol{\theta}}(\mathbf{x})$ son funciones de los parámetros $\boldsymbol{\theta}$ y serán considerados para aproximar f.

Definición 2.1.3. Sean $\mathcal{D}$ y $\mathcal{H}$ los conjuntos de datos y de hipótesis, respectivamente. Llamamos FUNCIÓN ERROR, FUNCIÓN DE COSTO (o *loss*) a

$$\mathcal{E}(\mathbf{Y}, h_{\boldsymbol{\theta}}(\mathbf{X})),$$

que permite cuantificar qué tan bien aproxima h a la función objetivo f, al comparar las salidas de ambas.

Definición 2.1.4. Sea $\mathcal{H}$ el conjunto de hipótesis y $\mathcal{E}$ la función error. Llamamos $\mathcal{A}$ al ALGORITMO DE APRENDIZAJE, que usa el conjunto de datos $\mathcal{D}$ para escoger la función $g \in \mathcal{H}$ que mejor aproxima la función objetivo, al buscar los parámetros $\boldsymbol{\theta}$ que mejor se ajustan a los datos, es decir, que minimizan la función error.

A pesar de lo que se espera, no usamos todos los datos etiquetados presentes en $\mathcal{D}$ para entrenar al algoritmo, sino que es conveniente apartar algunos ejemplos para después poner a prueba el modelo que mejor se ajuste al resto de los datos, usando estos ejemplos a los que no tuvo acceso durante el entrenamiento. Es por eso que se distinguen entre conjuntos de entrenamiento, validación y testeo.

Conjuntos de entrenamiento, validación y testeo

Para comenzar un análisis usando técnicas de AA lo primero que se lleva a cabo es una división aleatoria de los datos presentes en $\mathcal{D}$ en tres grupos mutuamente excluyentes $\mathcal{D}_{train}$, $\mathcal{D}_{val}$ y $\mathcal{D}_{test}$, llamados CONJUNTO DE ENTRENAMIENTO, VALIDACIÓN Y TESTEO, respectivamente [76, 80]. En general el primero de éstos tiene el mayor número de datos (por ejemplo, el 80 % ó 90 %), mientras que el resto se divide entre el segundo y tercer conjunto.

Luego de "alimentar" al algoritmo con los datos de $\mathcal{D}_{train}$, éste ajusta los parámetros $\boldsymbol{\theta}$ de forma tal que el error sea mínimo. El proceso de aprendizaje es iterativo: cada vez que se ingresa la totalidad de datos se dice que se ha completado una ÉPOCA (*epoch*) o experimento. Así, luego de la $n-$ésima época el modelo se ajusta minimizando la función error usando solamente los datos del conjunto de entrenamiento [80]:

$$\hat{\boldsymbol{\theta}}_n = \arg\min_{\theta} \left\{ \mathcal{E}(\mathbf{Y}_{train}, h_{\boldsymbol{\theta}}(\mathbf{X}_{train})) \right\},$$

donde para un conjunto arbitrario X, un conjunto completamente ordenado Y y una función $g : X \to Y$, el arg mín sobre un subconjunto $S \subseteq X$ está definido por [43]:

$$\arg\min_{x \in S \subseteq X} g(x) = \{ x \in S \mid g(y) \geq g(x)\ \forall y \in S \}.$$

A su vez durante el entrenamiento, luego de cada época, se valida el modelo usando los datos de $\mathcal{D}_{val}$, teniendo en cuenta los valores de los parámetros ajustados para ese experimento. A diferencia del proceso anterior, durante la etapa de validación no se

vuelven a ajustar los parámetros. El conjunto de validación permite evaluar qué tan bien el modelo generaliza durante el entrenamiento al calcular $\mathcal{E}(\mathbf{Y}_{val}, h_{\hat{\boldsymbol{\theta}}_n}(\mathbf{X}_{val}))$. Luego se selecciona el modelo cuyos parámetros $\hat{\boldsymbol{\theta}}$ hagan menor el error anterior. Finalmente, el rendimiento del modelo elegido se evalúa calculando la función de costo usando $\mathcal{D}_{test}$, esto es $\mathcal{E}(\mathbf{Y}_{test}, h_{\hat{\boldsymbol{\theta}}}(\mathbf{X}_{test}))$ [80].

En el caso en el que el conjunto de datos sea muy grande y no se pueda ingresar en un solo paso, se lleva a cabo el entrenamiento por mini lotes. Luego, cuando finaliza la época se valida el modelo. El TAMAÑO DEL LOTE (*batch size*) es un hiperparámetro que representa el número de ejemplos de entrenamiento presente en un solo lote, mientras que el número de lotes (*number of batches*) necesarios para completar una época se conoce como iteración. Por ejemplo, si se tienen 2000 ejemplos de entrenamiento, se pueden armar lotes con un tamaño de 500 ejemplos por lo que se necesitan 4 iteraciones para completar un experimento.

Teniendo en cuenta los conjuntos antes mencionados, definimos error dentro y fuera de la muestra [80, 76].

Definición 2.1.5. Llamamos ERROR DENTRO DE LA MUESTRA al valor de la función de costo para el modelo que mejor se ajusta usando el conjunto de entrenamiento,

$$E_{dentro} = \mathcal{E}(\mathbf{Y}_{train}, h_{\boldsymbol{\theta}}(\mathbf{X}_{train})),$$

mientras que el valor de la función de costo sobre el conjunto de testeo se llama ERROR FUERA DE LA MUESTRA O ERROR DE GENERALIZACIÓN,

$$E_{fuera} = \mathcal{E}(\mathbf{Y}_{test}, h_{\boldsymbol{\theta}}(\mathbf{X}_{test})).$$

En general, el error fuera de la muestra es mayor que el error dentro de la muestra, es decir, $E_{fuera} \geq E_{dentro}$. En los problemas de estadística clásica se suele partir de un modelo matemático que se supone verdadero y el objetivo es estimar el valor de algunos parámetros desconocidos. En cambio, los problemas de AA suelen implicar inferencias sobre sistemas complejos en los que no se conoce la forma exacta del modelo matemático que describe el sistema. Por lo tanto, lo usual es que se tengan múltiples modelos que describan al sistema que deban ser comparados [80]. Esta comparación se realiza mirando el error $\mathcal{E}(\mathbf{Y}_{val}, h_{\boldsymbol{\theta}}(\mathbf{X}_{val}))$: el modelo que lo minimiza se elige como el mejor modelo. Finalmente se lo evalúa en el conjunto de testeo para obtener una estimación de E_{fuera}.

El objetivo de un buen modelo de AA es generalizar bien desde los datos de entrenamiento a cualquier dato del dominio del problema, ya que esto permite hacer predicciones a futuro sobre datos que el modelo nunca ha visto. Sin embargo existen dos tipos de problemas, conocidos como sobreajuste y falta de ajuste [76, 72].

Sobreajuste y falta de ajuste

El SOBREAJUSTE (*overfitting*) se produce cuando un modelo aprende muy detalladamente de los datos de entrenamiento. Es decir, el ruido o las fluctuaciones aleatorias en los

datos de entrenamiento son aprendidos como conceptos por el modelo. El problema de ello es que estos conceptos no aplican a los nuevos datos y dificultan la capacidad del modelo de generalizar. Por otro lado, la FALTA DE AJUSTE (*underfitting*) se produce cuando el modelo no logra aprender de los datos de entrenamiento, ni logra generalizar a nuevos datos. En la Figura 2.2 podemos ver la diferencia entre un buen ajuste y los casos de falta de ajuste o sobreajuste para las tareas de clasificación y regresión, y cómo es el comportamiento para el caso de aprendizaje profundo, el cual presentaremos más adelante.

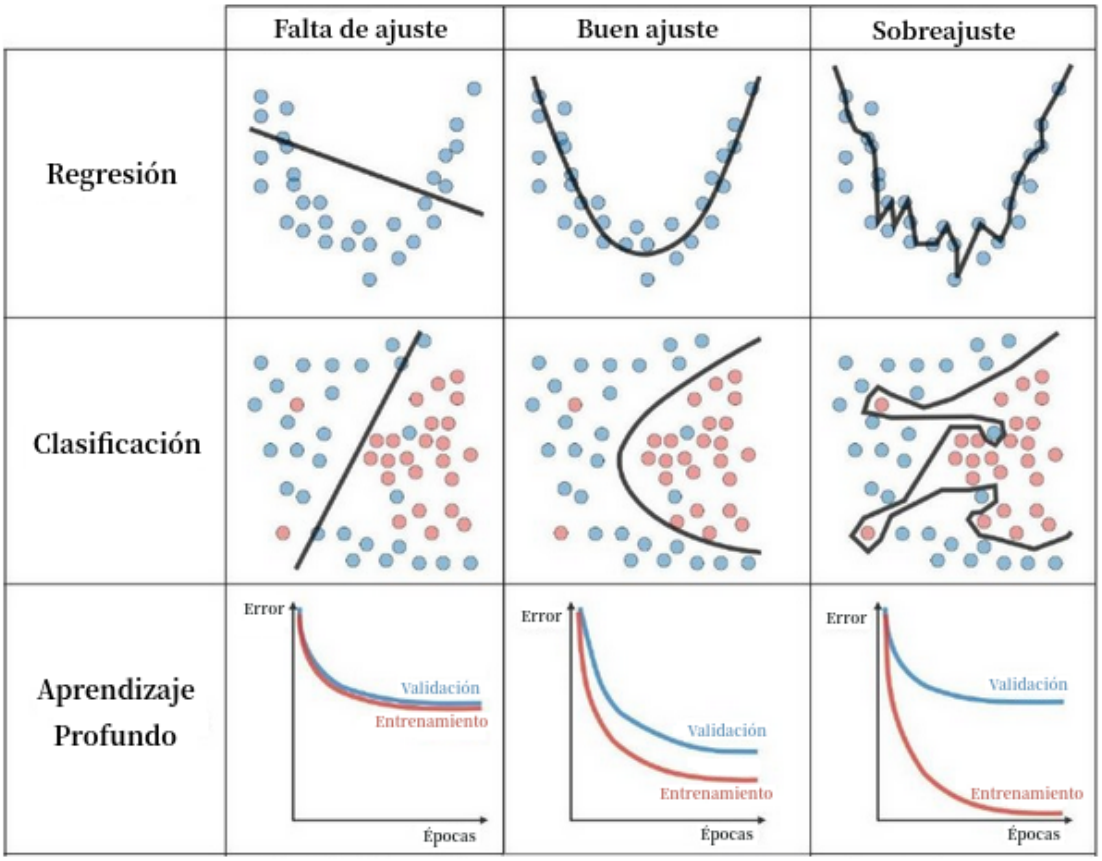


Figura 2.2. Comparación de un buen ajuste contra la falta de ajuste o sobreajuste.[2]

El tema de interés central en este trabajo es la detección de objetos en imágenes, la cual implica tareas de regresión y clasificación, por lo que en la siguiente Sección daremos una descripción de estas dos, así como las respectivas funciones de costo y distintas medidas de rendimiento.

[2]Imagen adaptada de [95].

2.2. Regresión y Clasificación

Las principales tareas enmarcadas en los algoritmos de aprendizaje supervisado son las de regresión y clasificación.

2.2.1. Problemas de regresión

El ANÁLISIS DE REGRESIÓN es una técnica de modelización predictiva que busca la relación entre una variable dependiente (llamada variable objetivo) y una o más variables independientes (llamados predictores). Esta técnica consiste en ajustar una curva a los datos observados, de forma tal que la diferencia entre los puntos y la curva sea mínima [92, 35]. Se utiliza para los pronósticos, la modelización de series temporales y la determinación de la relación causa-efecto entre variables.

Sea $N \in \mathbb{N}$. En los problemas de regresión la función $h : \mathcal{X} \subseteq \mathbb{R}^N \to \mathcal{Y} \subseteq \mathbb{R}$ que mejor aproxima la función objetivo f, predice un valor numérico $\hat{y} = h(\mathbf{x})$ dado un conjunto de características $\mathbf{x}$ (predictores). Las regresiones más comunes son las lineales: el modelo seleccionado es lineal en los parámetros, es decir, ningún parámetro en el modelo aparece como un exponente o multiplicado o dividido por otro parámetro. Si $N = 1$ se llama regresión lineal simple, caso contrario se llama regresión lineal múltiple.

En general, el modelo de REGRESIÓN LINEAL MÚLTIPLE con N variables independientes es de la forma

$$Y = b_0 + \sum_{i=1}^{N} b_i x_i + \epsilon,$$

donde ϵ es una variable aleatoria que se supone normalmente distribuida con $E(\epsilon) = 0$ y $Var(\epsilon) = \sigma^2$, y los parámetros b_i $(i = 0, \ldots, N)$ son los coeficientes de regresión. El parámetro b_j $(j \neq 0)$ representa el cambio esperado en Y por unidad de cambio en x_j, mientras el resto de las variables independientes x_i $(i \neq j)$ se mantienen constantes. La variable ϵ se conoce como término de error aleatorio o desviación aleatoria en el modelo [26].

En el caso en que una o más variables intervengan en una forma no lineal no debe considerarse como un modelo de regresión no lineal, ya que denominamos a un modelo de regresión como "lineal" cuando es lineal en los parámetros. Así, para el caso de relaciones no lineales entre las variables, también se usan regresiones lineales simples o múltiples (según corresponda) entre las transformaciones de los datos.

En estos problemas, la función de costo que se suele minimizar es la llamada ERROR CUADRÁTICO MEDIO (MSE, por sus siglas en inglés *Mean Square Error*) [76]. Si se tienen n observaciones, entonces:

$$\mathcal{E}_{MSE}(\mathbf{y}, h_{\boldsymbol{\theta}}(\mathbf{X})) = \frac{1}{n} \sum_{\mu=1}^{n} \left(y^{\mu} - \hat{y}^{\mu}\right)^2,$$

donde $\hat{y}^\mu = h_{\boldsymbol{\theta}}(\mathbf{x}^\mu)$ es el valor de predicción para una observación $\mathbf{x}^\mu$, e y^μ el valor verdadero observado, con $\mu = 1, 2, \ldots, n$. Por otro lado, otra función de costo posible es la SUMA DE LOS CUADRADOS DEL ERROR (SSE, por sus siglas en inglés *Sum of Squared Error*) [26]: $\mathcal{E}_{SSE} = n\,\mathcal{E}_{MSE}$.

Por ejemplo, para el caso $N = 1$ con variables lineales, los parámetros $\boldsymbol{\theta}$ a estimar son la pendiente y la ordenada al origen de la recta que relaciona x con y. Si bien este es un problema ampliamente estudiado en estadística, y son bien conocidos los métodos para obtener estimadores de los parámetros, no es objetivo de este trabajo ahondar en esta resolución, sino indicar que muchos algoritmos supervisados abordan las regresiones usando técnicas de AA como lo es el Gradiente Descendente (en especial cuando se tiene una gran cantidad de datos), el cual será explicado más adelante para el caso particular de Redes Neuronales.

2.2.2. Problemas de clasificación

Al igual que los modelos predictivos de regresión, los problemas de CLASIFICACIÓN son otro ejemplo de aprendizaje supervisado [76, 57]. Entre ellos se encuentran la clasificación binaria (*binary classification*) en el cual el algoritmo identifica si una única clase está o no está presente en cada ejemplo o muestra, mientras que en la clasificación multi-clase (*multi-class classification*) cada ejemplo pertenece a sólo una de C clases diferentes, y dos clases distintas no pueden estar presentes en la misma muestra. Por último, en la clasificación multi-etiqueta (*multi-label classification*) cada muestra puede pertenecer a más de una de las C clases, por lo que esta tarea se suele plantear como C problemas de clasificación binaria independientes.

Para los problemas de clasificación, como las etiquetas son categorías, se lleva a cabo una correspondencia de cada clase con un número diferente. Para clasificación binaria usualmente se usa 1 y 0 para indicar si está presente o no la clase en la muestra. Para clasificación multi-clase, como cada ejemplo sólo puede contener una única clase, se les asigna un orden a las C clases y, en caso que la muestra contenga la $i-$ésima clase de dicho orden, se usa un vector *one-hot* que tiene un 1 en la $i-$ésima posición y 0 en el resto. Para el caso de clasificación multi-etiqueta se usa un vector con tantos 1 como clases identificadas y 0 en el resto de las posiciones. Por esta particularidad de la tarea de clasificación, la función MSE no resulta apropiada para evaluar el error cometido. En cambio, se utiliza la función de costo de entropía cruzada (CE, por sus siglas en inglés *cross-entropy loss*). Para definir esta función error explicaremos su origen en la Teoría de la Información, para lograr un mayor entendimiento de la misma.

Teoría de la información y entropía

La teoría de la información [5] es una rama de la teoría de la probabilidad, que comenzó con Claude Shannon. Está relacionada con las leyes matemáticas que rigen la transmisión

y el procesamiento de la información y se ocupa de la medición de la información y de la representación de la misma, así como también de la capacidad de los sistemas de comunicación para transmitir y procesar información. Para esta teoría, la ENTROPÍA de una variable aleatoria es el nivel medio de "información", "sorpresa" o "incertidumbre" inherente a los posibles resultados de la variable.

La información que un suceso proporciona en sí mismo a un observador "objetivo" se conoce como su auto-información (*self-information*). La cantidad de auto-información $\mathrm{I}(\omega_n)$ contenida en un mensaje que transmite contenido informativo de un suceso ω_n depende solo de la probabilidad de ese evento, es decir,

$$\mathrm{I}(\omega_n) = f\left(\mathrm{P}(\omega_n)\right)$$

para alguna función continua $f(\cdot)$ que debe satisfacer [114]:

1. Si $\mathrm{P}(\omega_n) = 1$, entonces $\mathrm{I}(\omega_n) = 0$. Esto es, un evento con una probabilidad del 100 % no es nada sorprendente y no produce ninguna información.

2. Si $\mathrm{P}(\omega_n) < 1$, entonces $\mathrm{I}(\omega_n) > 0$. En particular, cuanto menos probable es un evento, más sorprendente es y más información produce.

3. Si A y B son sucesos independientes y $C = A \cap B$, entonces $\mathrm{I}(C) = \mathrm{I}(A) + \mathrm{I}(B)$. Esto es, si se miden dos eventos independientes por separado, la cantidad total de información de ambos sucesos a la vez es la suma de las auto-informaciones de los eventos individuales.

Como A y B son independientes, entonces de la condición 3 se desprende que

$$f\left(\mathrm{P}(A)\right) + f\left(\mathrm{P}(B)\right) = \mathrm{I}(A) + \mathrm{I}(B) = \mathrm{I}(C) = f\left(\mathrm{P}(C)\right) = f\left(\mathrm{P}(A) \cdot \mathrm{P}(B)\right)$$

La clase de funciones $f(\cdot)$ que satisfacen $f(x \cdot y) = f(x) + f(y)$ y la condición 1 es el logaritmo en cualquier base. Como la única diferencia entre logaritmos de diferentes bases es una constante de escala apropiada[3], entonces $f(x) = K \log_b(x)$. Por la condición 2 y dado que las probabilidades de los eventos son valores entre 0 y 1, entonces debe ser $K < 0$.

Teniendo en cuenta estas propiedades, la auto-información $\mathrm{I}(\omega_n)$ asociada al suceso ω_n de probabilidad $\mathrm{P}(\omega_n)$ se define como:

$$\mathrm{I}(\omega_n) = -\log_b\left(\mathrm{P}(\omega_n)\right) = \log_b\left(\frac{1}{\mathrm{P}(\omega_n)}\right),$$

donde en general $b = 2$, pero también son usuales la base e y la 10.

A partir de esto, se define la ENTROPÍA de una variable aleatoria discreta X que toma los valores posibles $\{x_1, x_2, \ldots, x_k\}$ y tiene función de probabilidad $\mathrm{P}(X)$ como

$$\mathrm{H}(X) = \mathrm{E}[\mathrm{I}(X)] = \mathrm{E}[-\log_b\left(\mathrm{P}(X)\right)],$$

[3]Sean $r = \log_b(x)$, $s = \log_b(k)$ y $t = \log_k(x)$, donde b, $k \in \mathbb{R}$. Entonces $b^r = x$, $b^s = k$ y $k^t = x$. Luego, $b^r = x = k^t = (b^s)^t = b^{st}$, es decir $\log_b(x) = r = st = \log_b(k)\log_k(x)$.

donde E es la esperanza ya que $\mathrm{I}(X)$ es una variable aleatoria. Así, explícitamente la entropía se calcula como[4]

$$\mathrm{H}(X) = -\sum_{i=1}^{k} \mathrm{P}(x_i) \log_b \left(\mathrm{P}(x_i)\right). \tag{2.1}$$

Entonces, se define la ENTROPÍA CRUZADA (*cross-entropy*) [76, 81, 57] de una variable aleatoria discreta X que toma los valores posibles $\{x_1, x_2, \ldots, x_k\}$, tiene función de probabilidad verdadera $\mathrm{P}(X)$ y función de probabilidad estimada $\mathrm{Q}(X)$ como

$$\mathrm{H}(\mathrm{P}, \mathrm{Q}) = \mathrm{E}_{\mathrm{P}}[-\log_b \left(\mathrm{Q}(X)\right)] = -\sum_{i=1}^{k} \mathrm{P}(x_i) \log_b \left(\mathrm{Q}(x_i)\right),$$

donde E_{P} es la esperanza respecto a la distribución P. La entropía cruzada entre dos distribuciones de probabilidad P y Q se interpreta como la incertidumbre esperada de una variable aleatoria distribuida según P, cuando se supone que es Q.

Clasificación

Volviendo a los problemas de clasificación, desde la estadística se usa el modelo de regresión logística para resolver tareas de clasificación binaria [26]. Este modelo establece una relación lineal entre la transformación logit y $N \in \mathbb{N}$ variables independientes, donde

$$\mathrm{logit}(p) = \ln\left(\frac{p}{1-p}\right), \text{ para } 0 < p < 1,$$

siendo p la probabilidad de éxito de la variable dependiente, la cual es una variable aleatoria Bernoulli. Esta relación lineal está dada por

$$\ln\left(\frac{p}{1-p}\right) = \theta_0 + \sum_{i=1}^{n} x_i \theta_i \tag{2.2}$$

Entonces, dada una observación $\mathbf{x}$, una vez hallados los parámetros $\boldsymbol{\theta}$ mediante el método de estimación de máxima verosimilitud, se calcula la probabilidad de clasificar dicha observación con éxito en la clase apropiada usando la función inversa de la ecuación (2.2), llamada función SIGMOIDE:

$$\sigma(\mathbf{x}) = p = \frac{1}{1 + e^{-(\theta_0 + \sum_{i=1}^{n} x_i \theta_i)}} \tag{2.3}$$

Y con un planteo similar se resuelven los problemas de clasificación multi-etiqueta y multi-clase.

[4] Shannon consideró la incertidumbre para una variable aleatoria y obtuvo directamente la función de entropía (2.1) en [2], sin definir la auto-información.

Maximizar la función de máxima verosimilitud en regresión logística es lo mismo que minimizar la función de costo de entropía cruzada (CE) en AA: si se tienen n observaciones y C clases, la CE es

$$\mathcal{E}_{CE}(\mathbf{y}, h_{\boldsymbol{\theta}}(\mathbf{X})) = \frac{1}{n}\sum_{\mu=1}^{n} \mathrm{H}(f,h) = -\frac{1}{n}\sum_{\mu=1}^{n}\sum_{c=1}^{C} f(\mathbf{x}^{\mu})^{c} \log_b \left(h_{\boldsymbol{\theta}}(\mathbf{x}^{\mu})^{c}\right), \tag{2.4}$$

donde f es la distribución de probabilidad verdadera (es decir, la distribución del conjunto de entrenamiento), y h la distribución descrita por el modelo.

En el caso de clasificación binaria, donde sólo se tienen la clase 0 y la clase 1, si f y h son como antes, el modelo asigna a cada entrada $\mathbf{x}$ la probabilidad de que la salida y esté en la clase 1 como $h(\mathbf{x}) = \mathrm{P}(y = 1|\mathbf{x}) = \hat{y}$, mientras que $\mathrm{P}(y = 0|\mathbf{x}) = 1 - \hat{y}$ es la probabilidad que pertenezca a la clase 0. Así, $f(\mathbf{x}) \in \{y, 1-y\}$ y $h_{\boldsymbol{\theta}}(\mathbf{x}) \in \{\hat{y}, 1-\hat{y}\}$ [116, 57, 81]. Entonces en este caso la ecuación (2.4) tiene la forma

$$\mathcal{E}_{Bin}(\mathbf{y}, h_{\boldsymbol{\theta}}(\mathbf{X})) = -\frac{1}{n}\sum_{\mu=1}^{n} \left[y^{\mu} \log_b \left(\hat{y}^{\mu}\right) + (1-y^{\mu}) \log_b \left(1-\hat{y}^{\mu}\right)\right].$$

Variaciones de la ecuación (2.4) se usan como función de costo para los problemas de clasificación multi-clase y multi-etiqueta. En particular, para este último se suman los errores de los C problemas binarios, y es lo que se conoce como *binary cross-entropy loss* [57]:

$$\mathcal{E}_{BCE}(\mathbf{y}, h_{\boldsymbol{\theta}}(\mathbf{X})) = \sum_{c=1}^{C} \mathcal{E}_{Bin}^{c}. \tag{2.5}$$

2.2.3. Medidas de rendimiento

Como su nombre lo indica, una medida de rendimiento es una forma de cuantificar qué tan bueno es un algoritmo, en el sentido de medir qué tan bien predice cuando se le presentan nuevos ejemplos a los que no tuvo acceso durante el entrenamiento. Como la naturaleza de los problemas no es siempre la misma, existen distintas medidas de rendimiento, dependiendo de si se trata de una regresión o una clasificación.

Regresión

Para determinar qué tan bien predice un modelo de regresión, se mide qué tan alejada está la predicción del valor que se espera obtener. Es por esto que tanto el MSE como la SSE sirven como medidas de rendimiento. También son usuales [76, 26]:

- Raiz Error Cuadrático Medio = $\sqrt{\mathcal{E}_{MSE}}$
- Error Absoluto = $\sum_{\mu=1}^{n} |y^{\mu} - \hat{y}^{\mu}|$

- Error Absoluto Medio $= \frac{1}{n}\sum_{\mu=1}^{n}|y^{\mu} - \hat{y}^{\mu}|$

- $R^2 = 1 - \frac{\mathcal{E}_{MSE}}{\sum_{\mu=1}^{n}(y^{\mu} - \bar{y})}$, donde $\bar{y} = \frac{1}{n}\sum_{\mu=1}^{n} y^{\mu}$

Clasificación

Una medida de rendimiento usual en clasificación binaria es la llamada MATRIZ DE CONFUSIÓN, y es una herramienta que ayuda a visualizar si el modelo se "confunde" al discriminar si el objeto pertenece a una clase A (lo que se dice caso positivo) o no (caso negativo). Al mirar la clase A, si un elemento es clasificado correctamente se dice Verdadero Positivo (VP), pero si es mal clasificado se dice Falso Negativo (FN). Un Falso Positivo (FP) es un elemento que no pertenece a A pero que es clasificado como si lo fuera, y un Verdadero Negativo (VN) es un elemento que no pertenece a la clase A correctamente clasificado. La matriz de confusión es una matriz de 2×2, cuyas filas representan las etiquetas reales de las clases y las columnas representan las clases predichas:

		Clases Predichas	
		Positivo	Negativo
Clases reales	Positivo	VP	FN
	Negativo	FP	VN

Así, los VP cuentan la cantidad de veces que el modelo clasifica correctamente casos positivos; los FN cuentan la cantidad de veces que el modelo clasifica incorrectamente un positivo como negativo; los FP la cantidad de veces que clasifica incorrectamente un negativo como positivo; y los VN la cantidad de veces que clasifica correctamente casos negativos. Para determinar si un objeto pertenece o no a A se compara alguna de sus características contra un valor umbral. Por lo tanto la matriz de confusión varía conforme se modifique dicho valor umbral. A partir de la matriz de confusión se pueden obtener las siguientes medidas de rendimiento [76].

La EXACTITUD (*accuracy*) describe el desempeño del modelo en todas las clases. Se calcula como el cociente entre el número de ejemplos clasificados correctamente y el total de las predicciones realizadas:

$$accuracy = \frac{\text{VP+VN}}{\text{VP+VN+FP+FN}}$$

La PRECISIÓN (*precision*) es calculada como el cociente entre el número de ejemplos positivos clasificados correctamente y el total de las clasificaciones positivas, ya sean correctas o incorrectas. Es decir, la precisión mide la exactitud del modelo para clasificar un ejemplo como positivo.

$$precision = \frac{\text{VP}}{\text{VP+FP}}$$

Cuando el modelo realiza muchas clasificaciones positivas incorrectas o pocas correctas, la precisión es pequeña. Esta medida de rendimiento refleja lo fiable que es el modelo para clasificar las muestras como positivas.

La EXHAUSTIVIDAD (*recall*) es el cociente entre el número de ejemplos positivos clasificados correctamente y el total de ejemplos positivos. La exhaustividad mide la capacidad del modelo de detectar ejemplos positivos.

$$recall = \frac{\text{VP}}{\text{VP+FN}}$$

Por lo tanto, la precisión mide la confianza del modelo en la clasificación de ejemplos positivos, y la exhaustividad mide cuántos ejemplos positivos fueron clasificadas correctamente por el modelo.

A mayor precisión, mayor confianza en el modelo al clasificar un ejemplo como positivo. A mayor exhaustividad, mayor cantidad de ejemplos positivos correctamente clasificados. Cuando un modelo tiene un alto grado de exhaustividad pero una baja precisión, entonces clasifica correctamente la mayoría de los ejemplos positivos pero tiene muchos falsos positivos (es decir, clasifica muchas muestras negativas como positivas). Cuando un modelo tiene una alta precisión pero baja exhaustividad, entonces es exacto cuando clasifica un ejemplo como positivo pero sólo puede clasificar unos pocos ejemplos positivos.

Habiendo introducido los principales conceptos relacionados con el aprendizaje supervisado, nos adentraremos ahora en un tipo de estructura en particular llamada Redes Neuronales, el cual será la base de nuestro trabajo. Veremos la inspiración biológica que le dio el nombre, así como su definición formal. Luego veremos cómo se comportan sus unidades (neuronas), definiendo las funciones intervinientes, para luego resumir esta información usando vectores y matrices. Así, una vez familiarizados con la estructura de la red, explicaremos en qué consisten el Gradiente Descendente así como la Retropropagación, procedimientos que son clave para comprender cómo aprende la red. Además veremos los principales problemas que surgen del entrenamiento, así como las posibles soluciones. Finalmente veremos en qué consiste el llamado aprendizaje residual.

2.3. Redes Neuronales

Una red neuronal artificial (ANN, por sus siglas en inglés *Artificial Neural Network*) es un modelo de computación inspirado en la estructura de las redes neuronales biológicas del cerebro. Se componen de un conjunto de unidades llamadas NEURONAS, organizadas en capas, que están conectadas entre sí en una red de comunicación compleja. Representan una de las técnicas de aprendizaje supervisado más poderosas y ampliamente utilizadas en la actualidad, ya sea para resolver problemas de regresión, como de clasificación [76].

2.3.1. Inspiración biológica

Una neurona biológica es una célula que se encuentra principalmente en las cortezas cerebrales de los animales, cuya función principal es recibir, procesar y transmitir información a través de señales químicas y eléctricas.

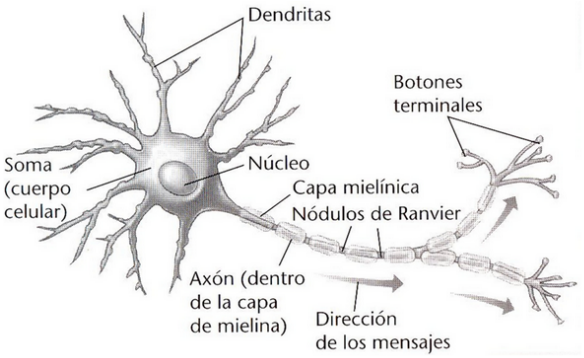
Las neuronas presentan características morfológicas típicas que son explicadas en numerosos libros, por ejemplo [15, 18], y se pueden observar en la Figura 2.3(a): un cuerpo celular, llamado soma, que contiene al núcleo; una o varias prolongaciones cortas que reciben los impulsos nerviosos de las neuronas adyacentes, denominadas dendritas; y una prolongación larga, denominada axón, que conduce los impulsos desde el soma hacia otras neuronas (o a músculos y glándulas). Cerca de su extremo el axón se divide en pequeñas ramificaciones, que terminan en botones llamados terminaciones sinápticas, que están conectadas a las dendritas de otras neuronas. Esta conexión no es física, ya que existe un pequeño espacio entre los botones terminales y el cuerpo celular o las dendritas de la neurona receptora. Esta unión se denomina sinapsis, y el espacio en sí se denomina espacio sináptico. Cuando un impulso nervioso viaja a través del axón y llega a los botones terminales, provoca la secreción de un neurotransmisor, una sustancia química que se difunde a través del espacio sináptico y estimula a la siguiente neurona, transmitiendo así el impulso de una neurona a otra. Dichos pulsos eléctricos se transmiten siempre que haya una diferencia de potencial significativa entre el interior y exterior del axón.

Si bien una neurona biológica parece comportarse de una manera bastante simple, lo cierto es que están organizadas en una vasta red de miles de millones de neuronas, donde cada una está conectada a miles de otras, como se aprecia en la Figura 2.3(b). Existen varios modelos que buscan imitar el comportamiento de las neuronas biológicas, de los cuales mencionamos a continuación uno de los más destacados.

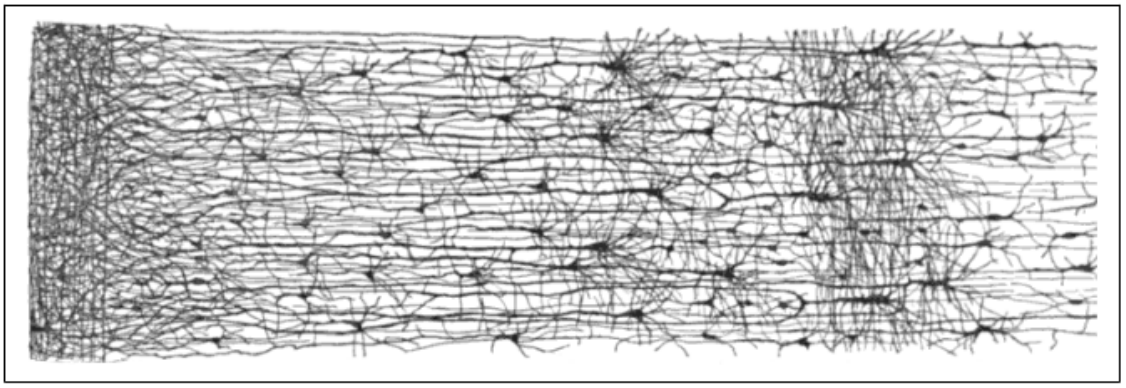
Neuronas artificiales

Una de los primeras neuronas artificiales que modeló una neurona biológica es la llamada NEURONA MCCULLOCH-PITTS [1] O UNIDAD LÓGICA DE UMBRAL (TLU, por sus siglas en inglés *Threshold Logic Unit*) [27]: si una neurona recibe una entrada excitatoria que no es compensada por una entrada inhibitoria igualmente fuerte, se activa y envía una señal a otras neuronas. Las entradas y salidas son números, y cada conexión tiene asociado un peso. La TLU calcula una suma ponderada de las entradas, y les aplica una función que tiene en cuenta un cierto valor umbral para determinar si la neurona dispara o no una señal. Una de las NN de arquitectura más simple es el llamado PERCEPTRÓN [4], el cual se basa en la TLU.

Una red neuronal artificial se compone de unidades de procesamiento simples llamadas neuronas, y de conexiones dirigidas y ponderadas entre esas neuronas. Formalmente, estos componentes se organizan en grafos, como veremos a continuación.



(a) Esquema de una neurona biológica.



(b) Corteza cerebral humana, donde se observan múltiples capas de neuronas de una red neuronal biológica.

Figura 2.3. Neuronas biológicas.[5]

2.3.2. Definición formal de NN

Sea X un conjunto. Notamos $\mathcal{P}_2(X) = \{S \subseteq X \mid |S| = 2\}$ al conjunto de partes de X formado por subconjuntos de dos elementos de X. Teniendo en cuenta esta notación, podemos describir una red neuronal artificial usando las siguientes nociones de Teoría de Grafos.

Definición 2.3.1. Sean N y A conjuntos.

- Un grafo es un par (N, A) donde $A \subseteq \mathcal{P}_2(N)$.
- Un grafo dirigido es un par (N, A) donde $A \subseteq N \times N$.

Llamamos vértices a los elementos de N, y aristas a los elementos de A. En un grafo las aristas son subconjuntos de dos elementos de N, mientras que en un grafo dirigido son pares ordenados (i, j), y decimos que la arista tiene una dirección de i a j.

Definición 2.3.2. Una función de peso en las aristas de un grafo se define como una función $w : A \to \mathbb{R}$, donde llamamos peso de la arista (i, j) a $w(i, j) = w_{i,j}$.

[5]Imágenes adaptadas de: (a) [108], (b) [76].

Entonces en base a lo anterior definimos red neuronal:

Definición 2.3.3. Una red neuronal es un grafo dirigido (N, A), donde N es finito y cuyas aristas son pares ordenados (i, j) tales que $i \neq j$, con una función de pesos sobre sus aristas. Llamamos neuronas o nodos a los vértices y conexiones entre neuronas a las aristas. En el caso que no existan conexiones entre neuronas se usa la convención $w_{ij} = 0$.

Cada conexión entre los nodos transmite una señal de una neurona a la otra: la neurona receptora procesa la señal y la envía a las neuronas conectadas a ella dentro de la red. A su vez estas conexiones tiene un peso asociado, el cual es un parámetro a aprender, que representa la fuerza de la conexión entre los dos nodos. Esta organización de las neuronas y sus conexiones es lo que describiremos en la próxima Sección. Inicialmente los pesos se eligen de forma aleatoria, y luego durante el entrenamiento se van actualizando al aplicar el descenso por el gradiente (Sección 2.3.4) a la función error que corresponda según el problema que se esté resolviendo.

2.3.3. Estructura

Según el modelo biológico, las neuronas se activan o excitan, y transmiten esa señal activando otras neuronas a su alrededor. Las reacciones de las neuronas a los valores de entrada dependen de su ESTADO DE ACTIVACIÓN.

Los nodos están organizados en lo que llamamos CAPAS, como se puede ver esquematizado en la Figura 2.4. Cada capa realiza un tipo de transformación diferente en sus datos según su activación. Dichos datos transformados fluyen a través de la red comenzando en la capa de entrada y avanzando a través de capas intermedias llamadas capas ocultas, hasta llegar a la capa de salida. Esto se conoce como propagación hacia adelante (*feed-forward*) a través de la red [18, 76]. A grandes rasgos, estas capas se diferencian unas de otras:

- Capa de entrada (*input layer*): tiene un nodo por cada componente o característica (predictor) de los datos de entrada. Es la única capa que no altera su estado de activación.

- Capas ocultas (*hidden layer*): se encuentran entre las capas de entrada y salida; el número de nodos en cada capa oculta es elegido arbitrariamente.

- Capa de salida (*output layer*): tiene un nodo Ω_l por cada una de las M componentes de la salida deseada, donde $l = 1, \ldots, M$.

Una capa oculta o de salida se dice COMPLETAMENTE CONECTADA o DENSA cuando todas sus neuronas están conectadas a cada una de las neuronas de la capa anterior, como las capas de la Figura 2.4. Existen además otros tipos de capas, por ejemplo las capas convolucionales (Sección 2.4.1) o de agrupamiento (Sección 2.4.3), entre otras. Un perceptrón simple es una NN con una capa de entrada y una capa de salida completamente

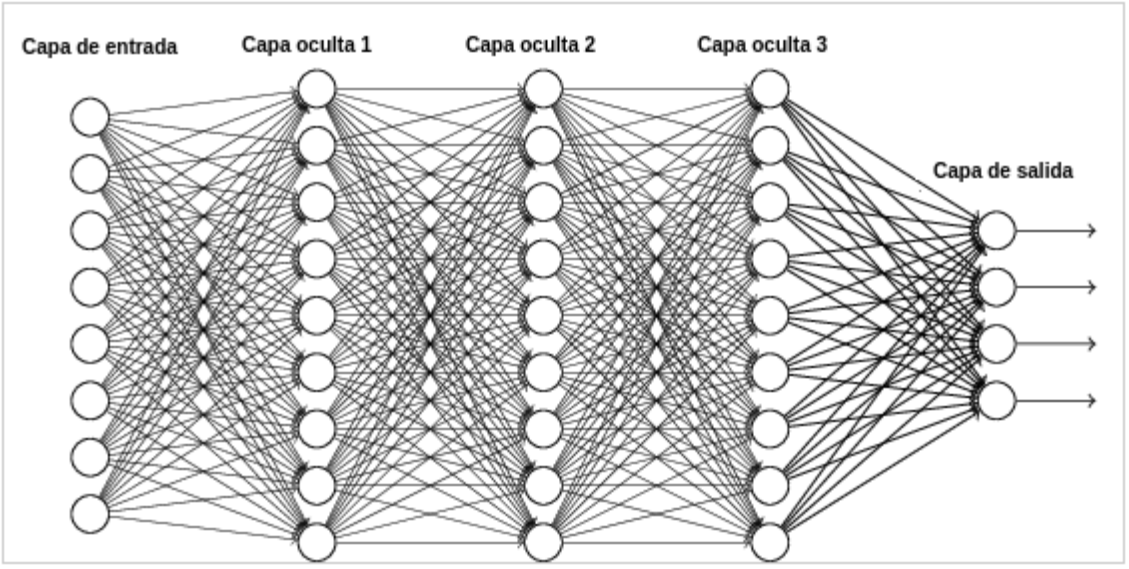


Figura 2.4. Estructura de una NN: las neuronas se organizan en capas.[6]

conectada, compuesta únicamente por TLUs. Por otro lado, un perceptrón multicapas (MLP por sus siglas en inglés *Multi-Layer Perceptron*) es una NN similar al perceptrón simple, que agrega varias capas ocultas de TLUs completamente conectadas [76]. En las próximas secciones describiremos cómo aprende una NN basándonos en la estructura del MLP, sin embargo estos conceptos se pueden generalizar para otro tipo de NN.

Cuando la red recibe un predictor del conjunto de entrenamiento en uno de los nodos de la capa de entrada, éste pasará al siguiente nodo a través de una conexión, para lo cual será multiplicado por el peso asignado a esa conexión. Para cada nodo de la segunda capa se calcula una suma ponderada teniendo en cuenta cada una de las conexiones entrantes, la cual representa el dato de entrada de una neurona de la segunda capa, y produce una salida que dependerá de la activación de dicha capa. Con estas salidas y con los pesos de conexión con la tercera capa, se calculan nuevas sumas ponderadas, y así sucesivamente. Cada nodo de las capas ocultas como de la capa de salida recibe datos de entrada, llamados ENTRADA DE CAPA y producen una salida, llamada SALIDA DE CAPA, que notamos z y o, respectivamente.

Sea $n \in \mathbb{N}$. Dada una neurona j se observan muchas neuronas conectadas a ésta, es decir, que transfieren sus salidas a j. Para esta neurona, existe una función de propagación que recibe las salidas $o_{i_1}, o_{i_2}, \ldots, o_{i_n}$ de las neuronas $i_1, i_2, \ldots, i_n$ (que están conectadas a j), y las transforma según los pesos de conexión w_{ij}, para luego ser mapeados por una función de activación. Así, la entrada de capa es el resultado de la función de propagación. En la Figura 2.5 se representa el esquema de una neurona, con las nociones antes descriptas, cuyas definiciones formales daremos a continuación siguiendo a [18].

[6]Imagen adaptada de [91].

Figura 2.5. Estructura de una neurona.[7]

Definición 2.3.4. Sea $p \in \mathbb{N}$. Sea $I = \{i_1, i_2, \ldots, i_p\}$ el conjunto de neuronas tales que para todo $m \in \{1, 2, \ldots, p\}$ existe $w_{i_m j}$, y sean $o_{i_1}, o_{i_2}, \ldots, o_{i_p}$ sus salidas. Entonces la ENTRADA DE CAPA de la neurona j, notada z_j, es calculada por la FUNCIÓN DE PROPAGACIÓN f_{prop} como

$$z_j = f_{prop}(o_{i_1}, o_{i_2}, \ldots, o_{i_p}, w_{i_1 j}, w_{i_2 j}, \ldots, w_{i_p j}).$$

Usualmente se usa la suma ponderada

$$z_j = \sum_{i \in I} o_i \, w_{ij}.$$

Si $I = \{x_1, x_2, \ldots, x_N\},\ N \in \mathbb{N}$, es el conjunto de neuronas de la capa de entrada, entonces la entrada de capa de la neurona j ubicada en la primer capa oculta está dada por $z_j = \sum_{i=1}^{N} x_i \, w_{ij}$.

Definición 2.3.5. Sea j una neurona. Definimos la FUNCIÓN DE ACTIVACIÓN f_{act} como la función que determina el estado de activación de la neurona j a partir de su valor de

[7] Imagen adaptada de [18].

entrada de capa, es decir,

$$a_j = f_{act}(z_j, \beta_j) = f_{act}\left(\sum_{i \in I} o_i \, w_{ij}, \beta_j\right),$$

donde β_j es el valor umbral de activación.

Existen distintos tipos de funciones de activación, por ejemplo en problemas de clasificación se suele usar la función sigmoide (Ecuación 2.3) en la capa de salida. En general cada neurona de una capa de una NN usa una misma función de activación, pero normalmente se utilizan diferentes funciones de activación en las diferentes capas, pudiendo repetirse alguna en particular. En la Figura 2.6 se definen y grafican las funciones de activación más usuales.

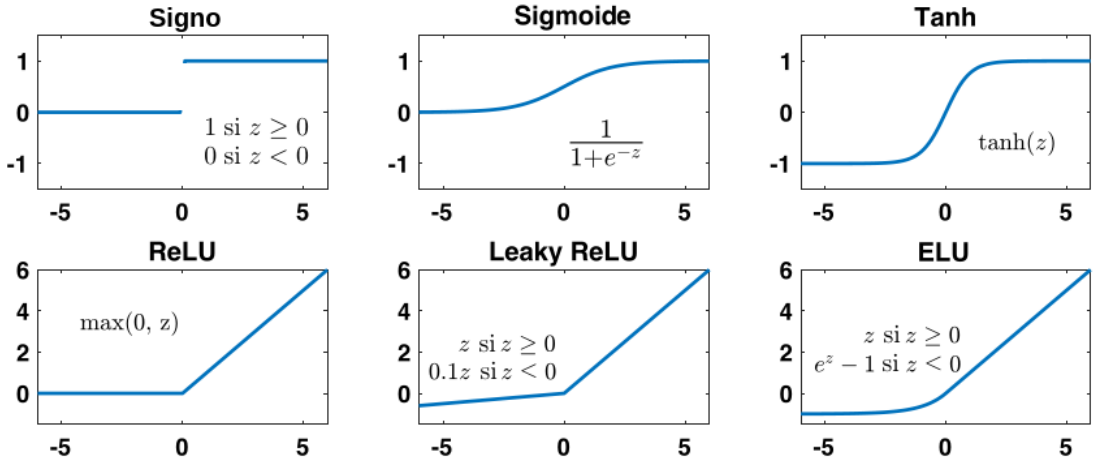


Figura 2.6. Funciones de activación.[8]

Como se muestra en la Figura 2.6, las funciones de activación definidas tienen $\beta_j = 0$ y en su mayoría satisfacen $f_{act}(0) = 0$. Para permitir una traslación horizontal se hace uso de un valor umbral de activación no nulo. Al igual que los pesos, dicho valor umbral representa un parámetro a aprender por la red, pero resulta complicado acceder a la función de activación durante el entrenamiento para aprender este parámetro.

Sin embargo los valores umbrales $\beta_{j_1}, \beta_{j_2}, \ldots, \beta_{j_p}$ para las neuronas $j_1, j_2, \ldots, j_p$ se pueden interpretar como los pesos de conexión de una neurona adicional integrada en la red y conectada a las neuronas $j_1, j_2, \ldots, j_p$, cuyo valor de salida es siempre 1, la cual se llama NEURONA DE SESGO (*bias neuron*).

Con esta incorporación, el valor umbral de las neuronas $j_1, j_2, \ldots, j_p$ se puede fijar en 0. Asi, los valores $\beta_{j_1}, \beta_{j_2}, \ldots, \beta_{j_p}$ se implementan como los pesos de conexión entre la neurona de sesgo y las neuronas $j_1, j_2, \ldots, j_p$, y pueden ser entrenados al mismo tiempo que los pesos w_{ij}, lo que facilita considerablemente el proceso de aprendizaje.

[8]Imagen adaptada de [80].

A partir de ahora, omitiremos la neurona de sesgo para mayor claridad en las fórmulas, pero no debe perderse de vista que existe y que los valores umbrales son tratados simplemente como pesos. Por lo tanto, todo lo referido a las conexiones w_{ij} es extensivo a los valores β_j, aunque no se escriba de forma explícita, y escribiremos la función activación simplemente como $a_j = f_{act}(z_j)$.

Definición 2.3.6. Sea j una neurona. La FUNCIÓN DE SALIDA

$$f_{out}(a_j) = o_j$$

calcula el valor de salida o_j de la neurona j según su estado de activación a_j, es decir o_j es la SALIDA DE CAPA de la neurona j.
En este trabajo usaremos la función identidad como función de salida, por lo que la activación a_j es directamente la salida, es decir,

$$f_{out}(a_j) = a_j \quad \text{i.e.} \quad o_j = a_j$$

Por otro lado, desde un punto de vista vectorial podemos definir:

Definición 2.3.7. Sean $N, M \in \mathbb{N}$. Dada una NN con N neuronas en la capa de entrada y M neuronas en la capa de salida, definimos el VECTOR DE ENTRADA $\mathbf{x} = (x_1, x_2, \ldots, x_N)$, de forma tal que la $i-$ésima componente x_i es la entrada del $i-$ésimo nodo de la capa de entrada. De forma análoga, llamamos VECTOR DE SALIDA a $\hat{\mathbf{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_M)$, donde $\hat{y}_l$ es la salida del $l-$ésimo nodo de la capa de salida, es decir, la salida de capa del nodo Ω_l es $o_l = \hat{y}_l$.

De esta forma se identifican la capa de entrada y de salida como los vectores de entrada y salida, respectivamente. La dimensión de entrada es N, y la dimensión de salida es M. Más aún, para n ejemplos presentes en el conjunto de entrenamiento, se organizan los datos en la matriz $\mathbf{X} \in \mathbb{R}^{n \times N}$, así como sus respectivas etiquetas $\mathbf{Y} \in \mathbb{R}^{n \times M}$.

Definición 2.3.8. Sean $n, N, M, K \in \mathbb{N}$. Dada una NN con K capas, N neuronas en la capa de entrada, M neuronas en la capa de salida, y un conjunto de entrenamiento de n ejemplos, definimos la MATRIZ DE PESOS con las conexiones entre capas sucesivas k y $k+1$ como

$$\mathbf{W}^{(k,k+1)} = \begin{pmatrix} w_{11}^{(k,k+1)} & w_{12}^{(k,k+1)} & \ldots & w_{1m_{k+1}}^{(k,k+1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m_k1}^{(k,k+1)} & w_{m_k2}^{(k,k+1)} & \ldots & w_{m_km_{k+1}}^{(k,k+1)} \end{pmatrix} \in \mathbb{R}^{m_k \times m_{k+1}},$$

donde m_k es la dimensión de la $k-$ésima capa, con $k = 1, 2, \ldots, K$ (por lo que $m_1 = N$ y $m_K = M$). Es decir, la entrada en la fila i, columna j de la matriz $\mathbf{W}^{(k,k+1)}$ está dada por el peso de conexión entre la neurona i de la capa k y la neurona j de la capa $k+1$.
Entonces, la entrada de la $k-$ésima capa ($k > 1$) está dada por

$$\mathbf{Z}^{(k)} = \mathbf{O}^{(k-1)} \cdot \mathbf{W}^{(k-1,k)},$$

donde $(\cdot)$ representa el producto matricial y $\mathbf{O}^{(k-1)} \in \mathbb{R}^{n \times m_{k-1}}$ es la matriz con las salidas de la capa $k-1$ para cada uno de los n ejemplos, usando la convención de notación $\mathbf{O}^{(1)} = \mathbf{X}$. Por otro lado, la salida de la $k-$ésima capa $(k > 1)$ está dada por

$$\mathbf{A}^{(k)} = f_{act}^{(k)}(\mathbf{Z}^{(k)}),$$

donde f_{act} se aplica a cada una de las componentes de $\mathbf{Z}$.

En lo que sigue nos concentraremos en describir cómo aprende una NN a partir del método de descenso por el gradiente. Como el objetivo es lograr entender el funcionamiento de estos métodos, la descripción será solamente para MLP, ya que es uno de los tipos de red más sencillos, sin embargo otro tipo de redes como las Convolucionales (Sección 2.4) también se basan en estos mismos conceptos.

2.3.4. Gradiente descendente

Donald Hebb sugirió en 1949 [3] que si dos neuronas están activas aproximadamente al mismo tiempo, la conexión entre ambas se vuelve más fuerte. Ésta se conoce como REGLA DE HEBB, y su interpretación es que el peso de conexión entre dos neuronas se incrementa siempre que tengan la misma salida. Los MLP se entrenan utilizando una variante de esta regla que tiene en cuenta el error cometido por la red, y refuerza las conexiones que ayudan a reducir el error.

Con esta regla podemos observar la relevancia que tienen los pesos w_{ij} en el proceso de aprendizaje, ya que determinan la importancia que tienen las conexiones. Recordemos que éstos son parámetros a aprender por el algoritmo: se quieren pesos grandes para conexiones fuertes, es decir, que contribuyan a llegar a la solución deseada, y pesos chicos para conexiones débiles, que no aportan a la solución deseada o que contribuyen equivocadamente, alejando de la solución esperada. El modelo será ajustado encontrando los pesos que minimizan la función error, la cual se mide a partir de las salidas obtenidas luego de ingresar los datos de entrenamiento a la NN.

Hallar los valores de los parámetros que minimizan la función error, la cual en principio se supone diferenciable, se lleva a cabo a partir de métodos de optimización numérica, ya que en general no existe una solución analítica. Para esto, se comienza en un valor inicial elegido aleatoriamente, y se repite una cantidad finita de operaciones, donde se actualiza dicho valor en cada paso o iteración. El método DESCENSO POR EL GRADIENTE es un algoritmo de optimización de primer orden, que se usa para encontrar un mínimo local de una función, que consiste en tomar pasos proporcionales al negativo del gradiente de la función en ese punto. Recordemos que [13]:

Definición 2.3.9. Sea $f : A \subseteq \mathbb{R}^n \to \mathbb{R}$ diferenciable, sea $\mathbf{a} = (a_1, a_2, \ldots, a_n) \in A$. El GRADIENTE de f en $\mathbf{a}$ es

$$\nabla f(\mathbf{a}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{a}), \frac{\partial f}{\partial x_2}(\mathbf{a}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{a}) \right)$$

Teorema 2.3.10. *Sea $f : A \subseteq \mathbb{R}^n \to \mathbb{R}$ diferenciable y $\mathbf{a} \in A$. Si $\nabla f(\mathbf{a}) \neq 0$, entonces $\nabla f(\mathbf{a})$ apunta en la dirección en la cual f crece más rápido, mientras que $-\nabla f(\mathbf{a})$ apunta en la dirección en la cual f decrece más rápido.*

Demostración. Sea $\mathbf{n}$ un vector unitario. Como f es diferenciable, entonces la derivada direccional de f en la dirección $\mathbf{n}$ está dada por

$$D_{\mathbf{n}} f(\mathbf{a}) = \nabla f(\mathbf{a}) \cdot \mathbf{n} = \|\nabla f(\mathbf{a})\| \|\mathbf{n}\| \cos\theta = \|\nabla f(\mathbf{a})\| \cos\theta,$$

donde θ es el ángulo entre $\mathbf{n}$ y $\nabla f(\mathbf{a})$. Por lo tanto, $D_{\mathbf{n}} f(\mathbf{a})$ es máximo cuando $\cos\theta = 1$ lo que implica que $\theta = 0$, esto es, cuando son vectores paralelos y de igual sentido. Por otro lado, es mínimo cuando $\cos\theta = -1$ lo que implica que $\theta = \pi$, esto es, cuando son vectores paralelos y de sentidos opuestos. □

Intuitivamente, como el gradiente apunta en la dirección de máximo crecimiento de una función, si el algoritmo toma un pequeño paso en la dirección contraria en cada iteración, decrecerán los valores de la función. En [30] se prueba que este método converge a un mínimo local de la función, así como se generaliza para funciones no diferenciables.

Para explicar en qué consiste el método, usaremos la notación que sigue a continuación. Sea $u^\mu = (\mathbf{x}^\mu, \mathbf{y}^\mu) \in \mathcal{D}_{train}$, con $\mu = 1, 2, \ldots, n$, tales que $\mathbf{x} \in \mathbb{R}^N$ e $\mathbf{y} \in \mathbb{R}^M$ (con $n, N, M \in \mathbb{N}$). Sean una familia $\mathcal{H}$ de funciones parametrizadas por un vector de pesos $\mathbf{w}$, y una función error $\mathcal{E}(\mathbf{y}^\mu, \hat{\mathbf{y}}^\mu) \in \mathbb{R}$, donde $\hat{\mathbf{y}}^\mu = h_{\mathbf{w}}(\mathbf{x}^\mu)$. Entonces se busca la función $h_{\mathbf{w}} \in \mathcal{H}$ que minimice $\mathcal{E}$ sobre toda la muestra $u^1, u^2, \ldots, u^n$.

El RIESGO EMPÍRICO $R_n(h)$ mide el rendimiento del conjunto de entrenamiento, mientras que el RIESGO ESPERADO $R(h)$ mide el rendimiento de la generalización [25], es decir, el rendimiento previsto en ejemplos futuros, donde

$$R_n(h) = \frac{1}{n} \sum_{\mu=1}^{n} \mathcal{E}(\mathbf{y}^\mu, h(\mathbf{x}^\mu)) \qquad \text{y} \qquad R(h) = \int \mathcal{E}(\mathbf{y}, h(\mathbf{x})) dP(u),$$

y es la teoría del aprendizaje la que justifica la minimización del riesgo empírico en lugar del riesgo esperado cuando la familia elegida $\mathcal{H}$ es suficientemente restrictiva, como se demuestra en [37]. El método propuesto para llevar a cabo dicha minimización es el algoritmo del descenso por el gradiente [25, 80]:

Definición 2.3.11. Para minimizar el riesgo empírico $R_n(h_{\mathbf{w}})$ se usa el método de DESCENSO POR EL GRADIENTE: en cada iteración se actualizan los pesos $\mathbf{w}$ en base al gradiente de $R_n(h_{\mathbf{w}})$. Si usamos la notación $\mathcal{E}(\mathbf{w}) = R_n(h_{\mathbf{w}})$, entonces

$$\Delta \mathbf{w}^t = -\eta \nabla \mathcal{E}(\mathbf{w})$$
$$\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t + \Delta \mathbf{w}^t$$

donde η es la TASA DE APRENDIZAJE (*learning rate*), $\mathbf{w}^t$ representa el peso en el tiempo actual, $\Delta\mathbf{w}^t$ representa la variación en el peso que se aplicará para actualizarlo en el siguiente paso de tiempo, es decir en $\mathbf{w}^{t+1}$; y la flecha $\leftarrow$ indica asignación.

La tasa de aprendizaje es un hiperparámetro a fijar, y su elección juega un rol fundamental en este algoritmo: si es demasiado pequeño, el descenso por el gradiente puede ser muy lento, pero si es muy grande podría no converger. Lamentablemente al tratarse de un hiperparámetro no hay seguridad de elegirlo correctamente, pero podemos guiarnos de la literatura para escoger un valor adecuado según el problema a resolver, o hacer varios intentos "a prueba y error". Por otro lado, si se comienza con un peso inicial $\mathbf{w}^0$ cerca del peso óptimo, el algoritmo anterior logra convergencia lineal, es decir, el error residual decrece exponencialmente, como se demuestra en [6].

En general, el descenso por el gradiente es un método que tiene varios limitantes, como se explica en [80], por ejemplo es muy sensible a la tasa de aprendizaje, así como a los valores iniciales, lo que podría llevar a encontrar un mínimo local no óptimo. Además es costoso en términos computacionales para conjuntos de datos numerosos. A continuación presentamos un método alternativo que soluciona estos limitantes.

Gradiente descendente estocástico

El DESCENSO POR EL GRADIENTE ESTOCÁSTICO (SGD, por sus siglas en inglés *Stochastic Gradient Descent*) [25, 80] en lugar de calcular el gradiente exacto de $R_n(h_{\mathbf{w}})$, lo estima en cada iteración usando un mini lote B_s de los datos.

Sean $n, q \in \mathbb{N}$. Si el conjunto de entrenamiento consta de n ejemplos y el tamaño de los lotes es $q < n$, entonces se trabaja con $\frac{n}{q}$ mini lotes. En general se pide que q divida a n, es decir $q \mid n$, pero si los tamaños no son los apropiados y existen c y r enteros no nulos tales que $n = c\,q + r$, entonces se trabaja con c mini lotes de tamaño q y un mini lote de tamaño r. En lo que sigue se asumirá que $q \mid n$, pero ténganse en cuenta que se puede trabajar con un lote de tamaño menor caso contrario.

Se ingresan a la NN los ejemplos de todos los mini lotes, de a un lote por vez, y se usa la aproximación del gradiente de cada mini lote para actualizar los parámetros $\mathbf{w}$ en cada paso $s = 1, 2, \ldots, \frac{n}{q}$. Recordemos que una época se completa cuando se usan los n ejemplos, es decir luego de pasar por los $\frac{n}{q}$ mini lotes.

Definición 2.3.12. Sean B_s los mini lotes, con $s = 1, 2, \ldots, \frac{n}{q}$. Sea

$$\nabla\mathcal{E}^{MB} = \frac{1}{q}\sum_{\mu\in B_s} \nabla\mathcal{E}(\mathbf{y}^\mu, h_{\mathbf{w}^{t,s}}(\mathbf{x}^\mu))$$

el cálculo del gradiente basado en el mini lote B_s. Teniendo en cuenta la notación de la definición 2.3.11, definimos el método de DESCENSO POR EL GRADIENTE ESTOCÁSTICO

(SGD):

$$\Delta \mathbf{w}^{t,s} = -\eta \nabla \mathcal{E}^{MB}$$
$$\mathbf{w}^{t+1} \leftarrow \mathbf{w}^{t} + \Delta \mathbf{w}^{t,s}$$

donde $\Delta \mathbf{w}^{t,s}$ representa la variación en el peso que se aplicará en la actualización, la cual tiene en cuenta el gradiente calculado en el mini lote B_s.

A continuación mencionamos un componente adicional que ayuda a mejorar el método del descenso por el gradiente estocástico.

Adición de momento

El descenso por el gradiente con momento es un método que introduce un término que recuerda lo ocurrido en la iteración anterior, por lo que la actualización de los parámetros usa una combinación lineal de los gradientes actual y anterior [25, 80].

Definición 2.3.13. Teniendo en cuenta la notación de la definición 2.3.12, definimos el método de DESCENSO POR EL GRADIENTE CON MOMENTO:

$$\Delta \mathbf{w}^{t,s} = \gamma \Delta \mathbf{w}^{t-1,s-1} - \eta \nabla \mathcal{E}^{MB}$$
$$\mathbf{w}^{t+1} \leftarrow \mathbf{w}^{t} + \Delta \mathbf{w}^{t,s}$$

donde $\Delta \mathbf{w}^{t-1,s-1}$ representa la variación en el peso aplicada en la iteración anterior (para un mini lote anterior) y γ es llamado PARÁMETRO DE MOMENTO, el cual satisface que $0 \leq \gamma \leq 1$. Así, cuando $\gamma = 0$ recuperamos la definición 2.3.12.

Habiendo introducido el método del descenso por el gradiente y algunas de sus modificaciones, presentamos ahora el algoritmo encargado del aprendizaje de la NN.

2.3.5. Entrenamiento: propagación hacia atrás

El algoritmo de propagación hacia atrás o retropropagación (*backpropagation*) [7] es esencialmente un SGD, diseñado de forma eficiente para calcular las actualizaciones de los pesos de todas las conexiones para reducir el error.

Para llevar a cabo la retropropagación se asume que ya se ha completado la propagación hacia delante. Es decir, luego de ingresar cada ejemplo de un mini lote a la red, se calculan la entrada y salida de capa en las capas ocultas y de salida, teniendo en cuenta los pesos de las conexiones, los datos ingresados y las funciones de activación [80, 107].

Sean $n, q, K, N, M \in \mathbb{N}$. Sea un MLP de K capas completamente conectadas, usada para entrenar con n ejemplos y con $\frac{n}{q}$ mini lotes de tamaño q. Sea $w_{ij}^{(k-1,k)}$ el peso de la

conexión del nodo i de la capa $k-1$ al nodo j de la capa k. Entonces, la entrada de capa de la neurona j es

$$z_j^{(k)} = \sum_{i=1}^{m_{k-1}} a_i^{(k-1)} w_{ij}^{(k-1,k)},$$

donde $m_k \in \mathbb{N}$ es la dimensión de la $k-$esima capa, con $k = 1, 2, \ldots, K$ (por lo que $m_1 = N$ y $m_K = M$), y $a_i^{(k-1)} = f_{act}^{(k-1)}\left(z_i^{(k-1)}\right)$. Sea $\mathcal{E}_l^\mu = \mathcal{E}(y_l^\mu, \hat{y}_l^\mu)$ el error cometido en la neurona Ω_l de la capa K, para el ejemplo μ. Sea B_s un mini lote ($s = 1, 2, \ldots, \frac{n}{q}$). Entonces el error cometido por la red luego de ver los q ejemplos del mini lote B_s es

$$\mathcal{E} = \sum_{\mu \in B_s} \sum_{l=1}^{M} \mathcal{E}(y_l^\mu, \hat{y}_l^\mu,),$$

siendo

$$\hat{y}_l^\mu = a_l^{(K),\mu} = f_{act}^{(K)}\left(z_l^{(K),\mu}\right) = f_{act}^{(K)}\left(\sum_{j=1}^{m_{K-1}} a_j^{(K-1),\mu} w_{jl}^{(K-1,K)}\right)$$

la salida obtenida, mientras que y_l^μ es la salida esperada (etiquetada). Por lo tanto, se observa que el error $\mathcal{E}$ es función de los pesos de todas las conexiones, no solo de las últimas conexiones w_{jl}, ya que cada estado de activación a_j se calcula teniendo en cuenta los pesos de conexión de la capa anterior.

Una vez obtenido $\mathcal{E}$ se procede a determinar la contribución de cada peso de conexión al error, para después aplicar SGD y actualizar los pesos, comenzando desde las conexiones entre la capa de salida y la última capa oculta. Así, para las neuronas j_0 y l_0 de las capas $K-1$ y K, respectivamente, por la regla de la cadena se tiene que:

$$\begin{aligned}\Delta w_{j_0 l_0}^{(K-1,K)} = -\eta \frac{\partial \mathcal{E}}{\partial w_{j_0 l_0}} &= -\eta \sum_{\mu \in B_s} \frac{\partial \mathcal{E}_{l_0}^\mu}{\partial \hat{y}_{l_0}} \cdot \frac{\partial \hat{y}_{l_0}}{\partial z_{l_0}^{(K)}} \cdot \frac{\partial z_{l_0}^{(K)}}{\partial w_{j_0 l_0}^{(K-1,K)}} \\ &= -\eta \sum_{\mu \in B_s} \frac{\partial \mathcal{E}_{l_0}^\mu}{\partial \hat{y}_{l_0}} \cdot f_{act}^{\prime(K)}\left(z_{l_0}^{(K)}\right) \cdot a_{j_0}^{(K-1)},\end{aligned}$$

donde se ha omitido el supra-índice μ que indica el número de ejemplo para facilitar la notación, excepto para $\mathcal{E}^\mu$. Por otro lado, como

$$a_j^{(K-1)} = f_{act}^{(K-1)}\left(z_j^{(K-1)}\right) = f_{act}^{(K-1)}\left(\sum_{i=1}^{m_{K-2}} a_i^{(K-2)} w_{ij}^{(K-2,K-1)}\right),$$

entonces para las neuronas i_0 y j_0 de las capas $K-2$ y $K-1$, respectivamente, se tiene que:

$$\begin{aligned}\Delta w_{i_0 j_0}^{(K-2,K-1)} = -\eta \frac{\partial \mathcal{E}}{\partial w_{i_0 j_0}} &= -\eta \sum_{\mu \in B_s} \sum_{l=1}^{M} \frac{\partial \mathcal{E}_l^\mu}{\partial \hat{y}_l} \cdot \frac{\partial \hat{y}_l}{\partial z_l^{(K)}} \cdot \frac{\partial z_l^{(K)}}{\partial a_{j_0}^{(K-1)}} \cdot \frac{\partial a_{j_0}^{(K-1)}}{\partial z_{j_0}^{(K-1)}} \cdot \frac{\partial z_{j_0}^{(K-1)}}{\partial w_{i_0 j_0}^{(K-2,K-1)}} \\ &= -\eta \sum_{\mu \in B_s} \sum_{l=1}^{M} \frac{\partial \mathcal{E}_l^\mu}{\partial \hat{y}_l} \cdot f_{act}^{\prime(K)}\left(z_l^{(K)}\right) \cdot w_{j_0 l}^{(K-1,K)} \cdot f_{act}^{\prime(K-1)}\left(z_{j_0}^{(K-1)}\right) \cdot a_{i_0}^{(K-2)}\end{aligned}$$

De forma análoga, se va propagando el error hacia atrás, hasta llegar a la capa de entrada. Para actualizar los pesos de las conexiones entre dos capas consecutivas $k-1$ y k se acumulan los cambios que se habrían hecho para cada ejemplo del lote. Así, se actualizan los pesos para cada capa ($k = 1, 2, \ldots, K$) y luego este procedimiento se repite para cada mini lote.

A medida que el algoritmo de propagación hacia atrás llega a las primeras capas, pueden darse ciertos problemas con las actualizaciones de los pesos, los cuales describimos a continuación, así como algunas posibles soluciones.

2.3.6. Deficiencias del gradiente descendente

A menudo, los gradientes se tornan muy pequeños a medida que el algoritmo de propagación hacia atrás llega a las capas inferiores. Como resultado, no se producen cambios significativos en los pesos de dichas capas durante la actualización del descenso por el gradiente, por lo que el entrenamiento no converge en una buena solución. Esto es lo que se conoce como PROBLEMA DE DESVANECIMIENTO DEL GRADIENTE (*vanishing gradients problem*). Por otro lado, los gradientes pueden aumentar cada vez más y llevar a actualizaciones muy grandes, por lo que el algoritmo diverge. Este es llamado PROBLEMA DE EXPLOSIÓN DEL GRADIENTE (*exploding gradients problem*). Estos problemas son comunes en las NN profundas (DNN por sus siglas en inglés *Deep Neural Networks*), compuestas por muchas capas ocultas, ya que las diferentes capas pueden aprender a velocidades muy diferentes [80, 76].

En la última década se han estudiado estos problemas, y propuesto diferentes soluciones, entre las cuales se destacan el uso de funciones de activación como la ReLU o la Leaky ReLU, que no saturan en los valores positivos (como sí lo hace la tangente hiperbólica o la sigmoide); y una nueva inicialización de los pesos, tomados de una distribución uniforme o una normal, cuyos parámetros (r o (μ, σ^2)) dependen del número de neuronas en las capas, en lugar de una normal estándar [21].

Normalización por lotes

Aunque los cambios en la función de activación e inicialización de los pesos reducen los problemas de desvanecimiento/explosión del gradiente al principio del entrenamiento, es posible que se vuelvan a originar más adelante durante el mismo. Para solucionar esto se utiliza la NORMALIZACIÓN POR LOTES (BN por sus siglas en inglés *Batch Normalization*) [33].

Este método consiste en agregar capas adicionales llamadas "BatchNorm" que normalizan las entradas por la media y varianza de los datos de los mini lotes. Dada una capa k de m_k neuronas, cuya entrada de capa es el vector $(z_1^{(k)}, z_2^{(k)}, \ldots, z_{m_k}^{(k)})$, se estandariza cada

componente haciendo

$$\hat{z}_j^{(k)} = \frac{z_j^{(k)} - \mu_{MB}}{\sqrt{\sigma_{MB}^2}},$$

donde la media μ_{MB} y la varianza σ_{MB}^2 son tomadas sobre todas los ejemplos del mini lote [80, 76].

Un problema de este procedimiento es que puede cambiar lo que la capa puede representar. Por ejemplo, en el caso de la tangente hiperbólica, puede llevar a que las salidas de capa estén limitadas a la linealidad existente en torno al cero. Para no perder la no linealidad, se introducen dos nuevos parámetros $\gamma_j^{(k)}$ y $\beta_j^{(k)}$ que desplazan y escalan la entrada normalizada

$$\hat{z}_j^{(k)} \leftarrow \gamma_j^{(k)} \hat{z}_j^{(k)} + \beta_j^{(k)},$$

donde los nuevos parámetros $\gamma_j^{(k)}$ y $\beta_j^{(k)}$ pueden ser aprendidos igual que los pesos, utilizando la propagación hacia atrás.

Para finalizar esta Sección, introduciremos el concepto de aprendizaje residual, conexiones de atajo y bloques residuales para el caso de MLP, como base para luego generalizarlo a Redes Convolucionales, ya que la estructura de red que usamos en este trabajo (YOLO), la cual veremos en detalle en la Sección 2.5.3, se caracteriza por tener este tipo de conexiones y bloques.

2.3.7. Aprendizaje residual

A pesar de lo que se creía, el apilar capas ocultas en una red no es equivalente a un mejor aprendizaje. Incluso solucionando los problemas de desvanecimiento/explosión del gradiente se comprobó empíricamente que las NN con muchas capas ocultas tenían mayor error de entrenamiento. Para solucionar esto se introdujo el APRENDIZAJE RESIDUAL (*residual learning*) [39].

Sea H la función objetivo a aprender luego de unas pocas capas apiladas (no necesariamente toda la red), y sea $\mathbf{x}$ la entrada a la primera de estas capas. Si se supone que $\mathbf{x}$ y $H(\mathbf{x})$ son de las mismas dimensiones, entonces en lugar de esperar a que el entrenamiento en dichas capas apiladas logre aproximar $H(\mathbf{x})$, se busca que se aproxime la FUNCIÓN RESIDUAL $F(\mathbf{x}) := H(\mathbf{x}) - \mathbf{x}$. Así la función original H se convierte en $F(\mathbf{x}) + \mathbf{x}$, que puede lograrse en los MLP con las llamadas CONEXIONES DE ATAJO (*skip connections* o *shortcut connections*), que se saltean una o más capas. En este caso, las conexiones de atajo aplican la función identidad y sus resultados se añaden a los resultados de las capas apiladas, como se muestra en la Figura 2.7. Se considera un BLOQUE RESIDUAL a

$$\mathbf{y} = F_{\mathbf{w}}(\mathbf{x}) + \mathbf{x},$$

donde $\mathbf{x}$ e $\mathbf{y}$ son los vectores de entrada y salida de las capas consideradas, y $F_{\mathbf{w}}(\mathbf{x})$ representa la función residual a aprender.

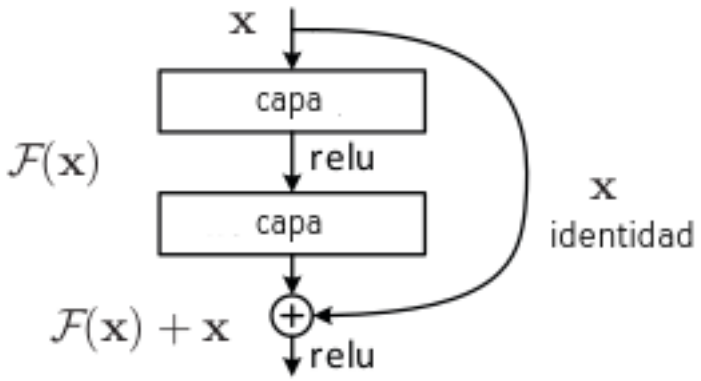


Figura 2.7. Aprendizaje residual.[9]

En el caso en que $\mathbf{x} \in \mathbb{R}^n$ y $F(\mathbf{x}) \in \mathbb{R}^m$, con $m > n$, se realiza una transformación lineal $f_{\mathrm{A}}(\mathbf{x}) = \mathrm{A}\mathbf{x}$, donde $\mathrm{A} \in \mathbb{R}^{m \times n}$ tiene por columnas los vectores canónicos $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n \in \mathbb{R}^m$.

Hasta ahora hemos representado la capa de entrada como un vector. En el caso que nuestra entrada fuese una imagen en escala de grises de m píxeles de alto y n píxeles de ancho, con $m, n \in \mathbb{N}$, su representación usual no es un vector, sino una matriz de tamaño $m \times n$, donde cada entrada se corresponde con la intensidad del píxel en esa posición. Entonces para poder ingresar la imagen a la red deberíamos hacer una transformación "aplanadora" para llevar los datos de la matriz a un vector de mn componentes, y si bien esto es posible de hacer tenemos la desventaja de perder la noción de entorno de cada píxel cuando realizamos dicha transformación. Por ejemplo, en una tarea de identificación de rostros en imágenes, es importante conocer entornos de los píxeles porque, por ejemplo, alrededor de los píxeles que cubren los ojos podrían existir lunares o pecas que pueden ayudar a identificar a la persona en cuestión, por lo que esa información se vuelve valiosa. En cambio, en una situación donde se "aplanó" la matriz, esta información podría quedar muy alejada de los píxeles que cubren los ojos.

Así, los problemas que involucran imágenes, entre otros, se resuelven usando una nueva estructura de NN, conocida como Red Neuronal Convolucional, de la cual hablaremos en la siguiente Sección.

[9]Imagen adaptada de [39].

2.4. Redes Neuronales Convolucionales

Las REDES NEURONALES CONVOLUCIONALES [11] (CNN, por sus siglas en inglés *Convolutional Neural Networks*) son un tipo de redes neuronales usadas especialmente para análisis de imágenes, ya que se caracterizan por detectar patrones y conservar la relación espacial de los datos de entrada. A diferencia de los MLP, las capas de las CNN tienen neuronas organizadas en tres dimensiones: ancho W, alto H y profundidad D (por ejemplo, las imágenes a color que ingresan a la red tienen profundidad 3, ya que se usa un canal RGB por cada color rojo, verde y azul), por lo que hablaremos de VOLUMEN DE ENTRADA Y SALIDA DE CAPA. Para almacenar esta información se usan TENSORES.

Existen varias formas de definir un tensor: como un arreglo multidimensional, como una transformación multilineal y como un elemento del espacio tensorial. Aunque aparentemente diferentes, los distintos enfoques describen el mismo concepto geométrico utilizando nomenclatura diferente en distintos niveles de abstracción. Aquí presentaremos la primera de éstas, que es la utilizada en AA [102].

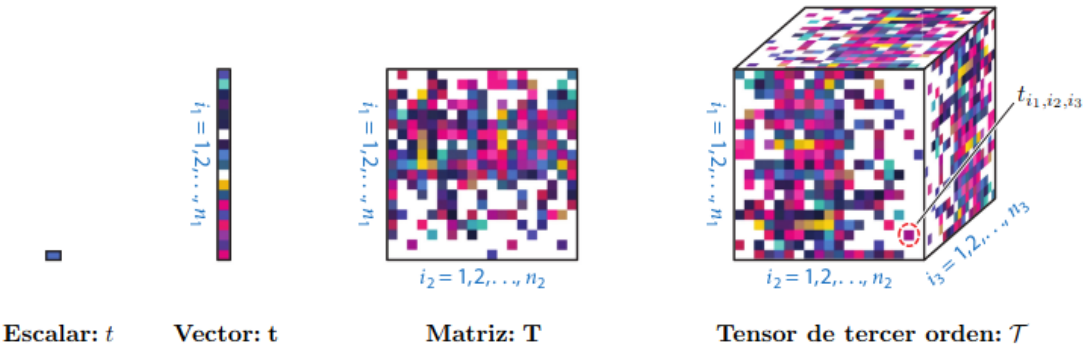
Definición 2.4.1. Un TENSOR de orden d es un arreglo $d-$dimensional, donde llamamos MODOS a sus dimensiones. Denotamos un tensor por $\mathcal{T} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$, donde n_j es la cantidad de componentes presente en el $j-$ésimo modo, con $j = 1, 2, \ldots, d$.

Un escalar $t \in \mathbb{R}$ es un tensor de orden 0 con una única componente, un vector $\mathbf{t} \in \mathbb{R}^n$ es un tensor de orden 1 con n componentes, y una matriz $\mathbf{T} \in \mathbb{R}^{n \times m}$ es un tensor de orden 2, con n componentes en el primer modo y m componentes en el segundo. Se denota por $t_{i_1,i_2,\ldots,i_d}$ para $i_j = 1, 2, \ldots, n_j$ y $j = 1, 2, \ldots, d$ al elemento $(i_1, i_2, \ldots, i_d)$ del tensor $\mathcal{T}$. Además, al fijar algunos índices del tensor, se definen sub-arreglos, por ejemplo una FIBRA (*fiber*) de un tensor es un vector que se crea fijando todos los índices excepto el de uno de los modos en particular. Así, un tensor $\mathcal{T}$ de tercer orden tiene fibras fila, columna y tubo, denotadas por $\mathbf{t}_{:,i_2,i_3}$, $\mathbf{t}_{i_1,:,i_3}$ y $\mathbf{t}_{i_1,i_2,:}$, respectivamente. Por otro lado, un CORTE (*slide*) de un tensor es una matriz bidimensional, definida fijando todos los índices menos los de dos modos particulares. Para el caso de un tensor $\mathcal{T}$ de orden 3, los cortes horizontal, lateral y frontal se denotan por $\mathbf{T}_{i_1,:,:}$, $\mathbf{T}_{:,i_2,:}$ y $\mathbf{T}_{:,:,i_3}$, respectivamente. En la Figura 2.8 están representados tensores de orden 0 a orden 3, así como los cortes de un tensor de tercer orden.

Para representar imágenes, se usan tensores de tercer orden de tamaño $W \times H \times D$ (con $W, H, D \in \mathbb{N}$), donde W y H son el número de píxeles en el ancho y alto de la imagen, respectivamente, y D es la profundidad ($D = 1$ para imágenes en escala de grises, y $D = 3$ para imágenes a color); mientras que se usan tensores de cuarto orden de tamaño $|B_s| \times W \times H \times D$ para representar una imagen de un mini lote, donde $|B_s|$ es el tamaño del mini lote.

Así como se usan tensores de tercer o cuarto orden para representar las imágenes que ingresan a la red en la capa de entrada, también se utilizan estos arreglos en el resto de las

capas de una CNN, en particular en las llamadas capas convolucionales.



(a)

(b)

Figura 2.8. Representación de tensores de distintos órdenes (a) y cortes de un tensor de tercer orden (b).[10]

2.4.1. Capas convolucionales

La arquitectura de las CNN se caracteriza por tener CAPAS OCULTAS CONVOLUCIONALES, que difieren de las capas densas en que cada neurona está conectada solamente a una región local (a lo largo del ancho y alto) del volumen de entrada de capa. El nombre de estas capas guarda relación con la operación de convolución (que denotamos con el símbolo $(*)$), aunque de hecho se trata de una operación de correlación cruzada (que denotamos con el símbolo $(\star)$) [14], las cuales definiremos a continuación. Los parámetros a aprender, es decir los pesos de las conexiones para dicha región local, son organizados en los llamados FILTROS (*filters*) o NÚCLEOS DE CONVOLUCIÓN (*convolution kernels*). Estas regiones son pequeñas en términos de ancho y alto, pero se extienden a través de la profundidad del volumen, y su tamaño se fija con un hiperparámetro.

[10] Imágenes adaptadas de [102].

Definición 2.4.2. Sean f y g funciones reales, continuas por tramos en $\mathbb{R}$. Definimos la CONVOLUCIÓN de f y g como:

$$(f * g)(t) = (f(\tau) * g(\tau))(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau.$$

Para el caso en que f y g tengan dominio en $\mathbb{Z}$, definimos la convolución como:

$$(f * g)(m) = (f(n) * g(n))(m) = \sum_{n} f(n)g(m-n).$$

Definición 2.4.3. Sean f y g funciones reales, continuas por tramos en $\mathbb{R}$. Definimos la CORRELACIÓN CRUZADA (*cross-correlation*) de f y g como:

$$(f \star g)(t) = (f(\tau) \star g(\tau))(t) = \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau.$$

Para el caso en que f y g tengan dominio en $\mathbb{Z}$, definimos la correlación cruzada como:

$$(f \star g)(m) = (f(n) \star g(n))(m) = \sum_{n} f(n)g(m+n).$$

Proposición 2.4.4. *La correlación cruzada de las funciones $f(\tau)$ y $g(\tau)$ es equivalente a la convolución de $f(-\tau)$ y $g(\tau)$. Esto es:*

$$(f(\tau) \star g(\tau))(t) = (f(-\tau) * g(\tau))(t)$$

A pesar de su nombre, las capas convolucionales no aplican la operación de convolución, sino la de correlación cruzada [101]. Los filtros son tensores de tamaño $f_W \times f_H \times D$, donde f_W y f_H son el ancho y alto del núcleo, respectivamente, y D es la profundidad (que coincide con la profundidad del volumen de entrada de capa). Las entradas de los filtros son los pesos de las conexiones, que varían entre cortes frontales [76, 80]. Como las imágenes son representadas por tensores de tercer orden, y cada corte frontal es una matriz de tamaño $W \times H$, donde cada entrada es el valor del píxel correspondiente, se aplica la correlación cruzada discreta en dos dimensiones entre los píxeles seleccionados de una imagen y los pesos de un filtro [14], esto es se superponen el $l-$ésimo corte frontal del filtro sobre el $l-$ésimo corte frontal de la entrada de capa y se suman las multiplicaciones lugar a lugar:

$$(f \star g)(m_1, m_2) = \sum_{n_1} \sum_{n_2} f(n_1, n_2)g(m_1 + n_1, m_2 + n_2)$$

Luego se suma el resultado obtenido para cada uno de los cortes frontales y se desplaza el filtro para realizar el mismo cálculo. Un ejemplo de esto se observa en la Figura 2.9, donde por simplicidad se asume $D = 1$. Así, dadas una entrada de 5×5 y un filtro de 3×3, se obtiene una salida de 3×3, donde sus entradas son el resultado de la suma de los productos lugar a lugar de la superposición entre el filtro y la entrada. Otro ejemplo (dinámico) se puede observar en [50][11], para el caso de un volumen de entrada de $5 \times 5 \times 3$

[11] Link directo al demo: `https://cs231n.github.io/assets/conv-demo/index.html`

y dos filtros de tamaños $3 \times 3 \times 3$, que producen un volumen de salida de tamaño $3 \times 3 \times 2$ (ya que se usan tamaño de paso $S = 2$ y relleno de ceros $P = 1$, los cuales explicaremos más adelante). La convolución es igual que la correlación, salvo que volteamos el filtro antes de correlacionar. Por ejemplo, la convolución de una imagen hipotética de 1D con el filtro $(3, 7, 5)$ es exactamente igual que la correlación con el filtro $(5, 7, 3)$, mientras que en el caso de la convolución 2D, volteamos el filtro tanto horizontal como verticalmente. Entonces computacionalmente esta diferencia no afecta al rendimiento del algoritmo, porque durante la etapa de entrenamiento se buscan los pesos adecuados. Añadir la operación de volteo simplemente se traduciría en que el algoritmo aprenda los pesos en diferentes celdas del núcleo. Por lo tanto, se puede omitir el volteo, por eso es que se trabaja con la correlación cruzada [14].

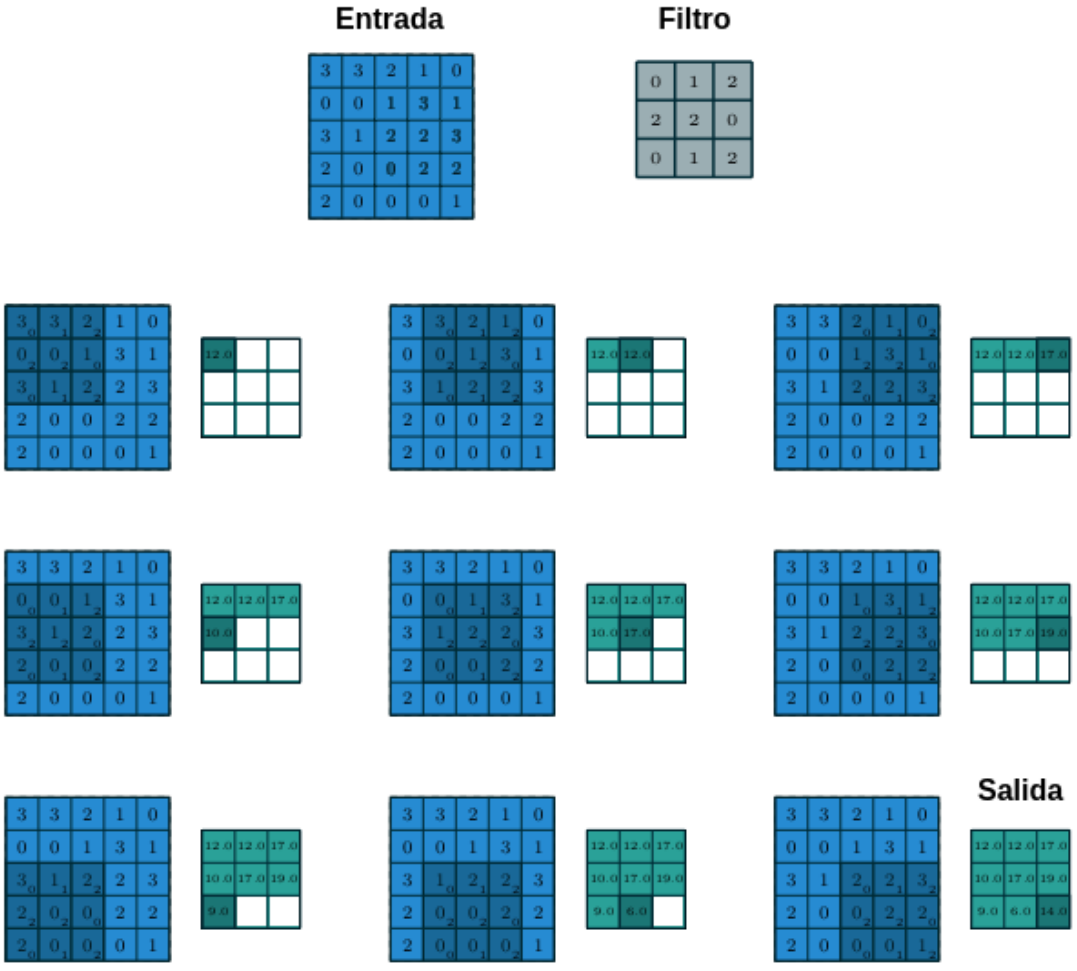


Figura 2.9. Operación de correlación cruzada entre una entrada 5×5 y un filtro 3×3. La salida obtenida es de 3×3.[12]

[12]Imagen adaptada de [38].

El CAMPO RECEPTIVO de una neurona (RF, por sus siglas en inglés *Receptive Field*) es una región local (incluyendo su profundidad) en el volumen de salida de la capa anterior a la que se conecta una neurona. Por otro lado, el CAMPO RECEPTIVO EFECTIVO (ERF, por sus siglas en inglés *Effective Receptive Field*) es el área de la imagen original que posiblemente puede influir en la activación de una neurona. El RF y el ERF son iguales para la primera capa convolucional, sin embargo difieren a medida que se avanza en las capas de la CNN, ya que el ERF indica qué área de la imagen de entrada es considerada por el filtro de las capas intermedias, y su cálculo se hace de forma recursiva [51].

Las neuronas de la primera capa convolucional no están conectadas a cada uno de los píxeles de la imagen de entrada, sino sólo a los píxeles de sus campos receptivos. Durante la propagación hacia adelante, se realiza la correlación cruzada de cada filtro a través del ancho y alto del volumen de entrada, y se produce un mapa de características bidimensional con los resultados de esta operación (por lo que hay tantos mapas de características como filtros aplicados), que resalta las áreas de la imagen que activan más el filtro. Al apilar dichos mapas a lo largo de la dimensión de profundidad se obtiene el volumen de salida. Todas las neuronas dentro de un mapa de características dado comparten los mismos parámetros (pesos), lo que reduce significativamente el número de parámetros del modelo, pero además las neuronas de los diferentes mapas de características utilizan distintos parámetros.

Esta arquitectura permite a la red concentrarse en pequeñas características de bajo nivel en las primeras capas ocultas. Intuitivamente, la red aprende filtros que se activan cuando ven algún tipo de característica específica, como un borde o mancha de algún color en las primeras capas, mientras que aprende patrones enteros como formas geométricas en las próximas capas, y objetos específicos como animales o plantas en las últimas capas. Una capa convolucional aplica simultáneamente múltiples filtros entrenables a sus entradas, lo que la hace capaz de detectar múltiples características en cualquier parte de sus entradas. No es necesario definir los filtros manualmente, sino que durante el entrenamiento la capa convolucional aprende automáticamente los filtros más apropiados para su tarea, mientras que las últimas capas aprenden a combinarlos en patrones más complejos [76].

El tamaño del volumen de salida de cada capa convolucional además de depender del tamaño del núcleo, está determinado por los hiperparámetros profundidad D, paso (*stride*) S_W y S_H y relleno de ceros (*zero padding*) P. El primero se corresponde con la cantidad de filtros usados por capa (si se trata de la capa de entrada entonces D es la cantidad de canales: $D = 1$ en imágenes en escala de grises, y $D = 3$ en imágenes a color); el segundo refiere al tamaño del paso con el que se desliza el filtro a lo ancho y alto, por ejemplo si es de tamaño 1 los filtros se mueven un píxel a la vez, pero si es de tamaño 2 entonces los filtros saltan dos píxeles a la vez a medida que son deslizados. Por último, es usual rellenar el volumen de entrada con ceros alrededor del borde, lo que permite controlar el tamaño del volumen de salida; el tamaño de dicho relleno es el tercer hiperparámetro [76, 50]. En la Figura 2.10 se observa el desplazamiento de un núcleo de tamaño 3×3 que se aplica a

un mapa de características con $D = 1$ para facilitar la representación. Se aplica un paso $S_W = S_H = 1$ y relleno de ceros $P = 1$.

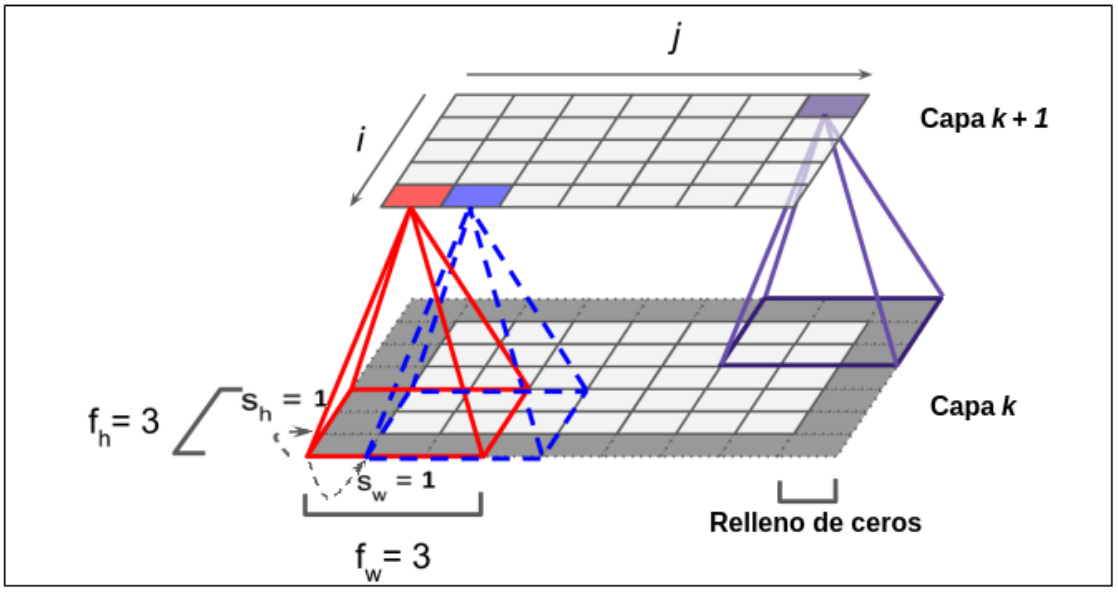


Figura 2.10. Desplazamiento de un núcleo, indicando tamaño de filtro, tamaño de paso y relleno de ceros.[13]

La $k-$ésima capa convolucional de $L^{(k)}$ filtros de tamaño $f_W^{(k)} \times f_H^{(k)}$, mismo tamaño de paso $S^{(k)}$ a lo ancho y alto, y tamaño de relleno de ceros $P^{(k)}$, cuyo volumen de entrada es de tamaño $W_1 \times H_1 \times D_1$, tendrá volumen de salida de tamaño $W_2 \times H_2 \times D_2$, donde $W_2 = (W_1 - f_W^{(k)} + 2P^{(k)})/S^{(k)} + 1$, $H_2 = (H_1 - f_H^{(k)} + 2P^{(k)})/S^{(k)} + 1$ y $D_2 = L^{(k)}$ [80, 50]. Para el cálculo de W_2 y H_2 se asume que la división es entera, si ese no fuera el caso los algoritmos computacionales en general agregan mayor cantidad de ceros en los bordes para que así sea.

Una neurona ubicada en la fila i, columna j del $l-$ésimo mapa de características de la $k-$ésima capa convolucional está conectada a las salidas de las neuronas de la capa anterior, ubicadas de la fila $(i-1)S_H+1$ a la $(i-1)S_H+f_H$ y de la columna $(j-1)S_W+1$ a la $(j-1)S_W+f_W$, a través de todos los mapas de características que están en la capa $k-1$ [76]. Así, todas las neuronas situadas en la misma fila i y columna j pero en diferentes mapas de características están conectadas a las salidas del mismo conjunto de neuronas de la capa anterior. La salida $o_{i,j,l}$ de una neurona ubicada en la fila i, columna j del mapa de características l de la capa convolucional k está dada por:

$$o_{i,j,l} = \sum_{u=1}^{f_H} \sum_{v=1}^{f_W} \sum_{l'=1}^{L_{k-1}} z_{i',j',l'} . w_{u,v,l',l} \qquad \text{con} \quad i' = (i-1)S_H + u, \quad j' = (j-1)S_W + v,$$

donde $z_{i',j',l'}$ es la salida de la neurona de la capa $k-1$, ubicada en la fila i', columna j' y mapa de características l' (o canal l' si la capa anterior es la capa de entrada), y $w_{u,v,l',l}$

[13]Imagen adaptada de [76].

es el peso de conexión entre cualquier neurona del mapa de características l de la capa k y la entrada ubicada en la fila u, columna v (relativo al campo receptivo) del mapa de características l'. En la Figura 2.11 se representan la capa de entrada de una imagen de color (por lo que $D = 3$ para esta capa) y dos capas ocultas convolucionales con múltiples mapas de características ($D > 3$).

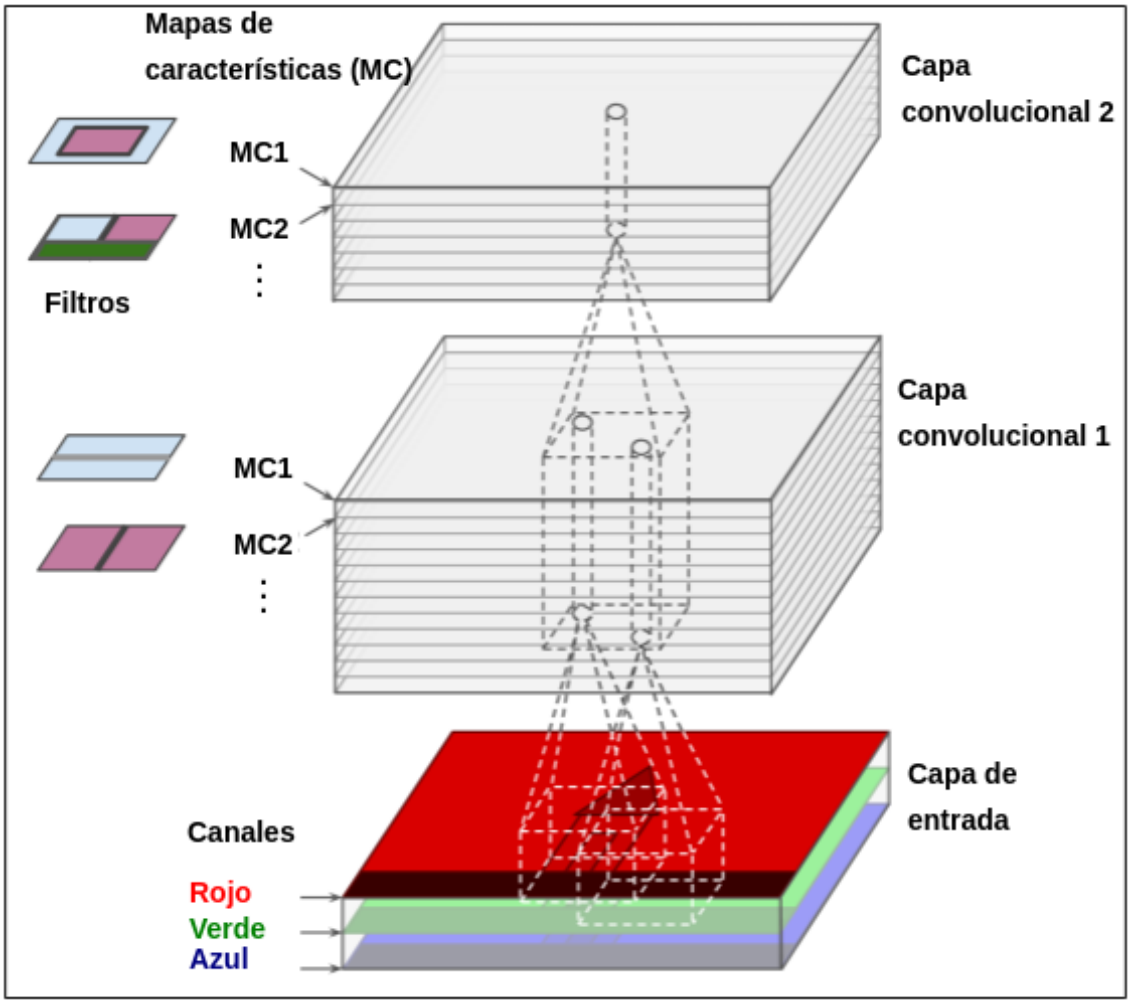


Figura 2.11. Capas convolucionales con múltiples mapas de características.[14]

A pesar de reducirse considerablemente el número de parámetros a aprender, las CNN tienen problemas de memoria, ya que requieren una gran cantidad de memoria RAM.

2.4.2. Requisitos de memoria

Las CNN requieren mucha memoria RAM especialmente durante el entrenamiento, porque para llevar a cabo la retropropagación se necesitan todos los valores de los parámetros calculados durante la propagación hacia adelante. Por ejemplo, consideremos una capa convolucional con filtros de 5×5, que produce 200 mapas de características de tamaño 150×100,

[14]Imagen adaptada de [76].

con paso $S = 1$ y relleno de ceros donde sea necesario. Si la entrada es una imagen RGB (tres canales) de 150×100, entonces el número de parámetros es $(5 \times 5 \times 3) \times 200 = 15000$, que es bastante pequeño comparado con una capa totalmente conectada.[15] Sin embargo, cada uno de los 200 mapas de características contiene 150×100 neuronas, y cada una de estas neuronas necesita calcular una suma ponderada de sus $5 \times 5 \times 3 = 75$ entradas: eso es un total de 225 millones de multiplicaciones de punto flotante[16]. Si bien es una menor cantidad de operaciones a realizar comparado con una capa totalmente conectada, sigue siendo muy exhaustivo desde el punto de vista computacional. Además, si los mapas de características se representan con puntos flotantes de 32 bits, la salida de la capa convolucional ocupará $200 \times 150 \times 100 \times 32 = 96$ millones de bits (12 MB) de RAM[17], siendo que esta estimación de memoria es sólo para una imagen. Si un lote de entrenamiento contiene 100 imágenes, entonces esta capa utilizaría 12 GB de RAM [76].

Cuando se realiza una predicción para una nueva imagen, la memoria RAM ocupada por una capa puede liberarse tan pronto como se haya calculado la siguiente capa, por lo que sólo se necesita la cantidad de RAM que requieren dos capas consecutivas. Pero durante el entrenamiento todo lo que se calcula durante la propagación hacia adelante debe conservarse para la retropropagación, por lo que la cantidad de RAM necesaria es, al menos, la cantidad total de RAM requerida por todas las capas.

Por otro lado, junto con las capas convolucionales se suelen encontrar las capas de agrupamiento, que ayudan a disminuir la cantidad de parámetros a aprender por la red. Como habíamos adelantado, en este trabajo usamos la arquitectura de la red YOLO, en particular su tercera versión, la cual explicaremos en detalle más adelante. Aunque ésta no utiliza capas de agrupamiento durante el entrenamiento, la segunda versión de YOLO es un ejemplo de NN que utiliza agrupamiento máximo.

2.4.3. Capas de agrupamiento

Otro tipo de capas que se encuentran comúnmente en las CNN son las llamadas CAPAS DE AGRUPAMIENTO (*pooling layers*) [76, 50], cuyo objetivo es encoger la imagen de entrada para reducir la carga computacional, el uso de memoria y el número de parámetros. Como en las capas convolucionales, cada neurona de una capa de agrupamiento está conectada a las salidas de un subconjunto de neuronas ubicadas en el campo receptivo de la capa

[15] Una capa densa con 150×100 neuronas, cada una conectada a las $150 \times 100 \times 3$ entradas, tendría $150^2 \times 100^2 \times 3 = 675$ millones de parámetros.

[16] La representación de punto flotante (*floating point*) es una forma de notación científica usada en las computadoras con la cual se pueden representar números reales extremadamente grandes y pequeños de una manera muy eficiente y compacta, y con la que se pueden realizar operaciones aritméticas.

[17] En el Sistema Internacional de Unidades (SI), 1 MB = 1000 kB = 1000×1000 bytes = $1000 \times 1000 \times 8$ bits.

anterior.

Para definir estas capas también se utilizan los hiperparámetros profundidad, paso y relleno de ceros. Sin embargo, a diferencia de las capas convolucionales, las capas de agrupamiento no tienen pesos. Lo que las caracteriza es que aplican una función de agrupación, como es el máximo o promedio. En el primer caso se toma el valor máximo entre las entradas de cada campo receptivo eliminando el resto de las entradas, mientras que en el segundo se toma el promedio de todas las entradas por campo receptivo. En la Figura 2.12 se observa un ejemplo de agrupamiento máximo.

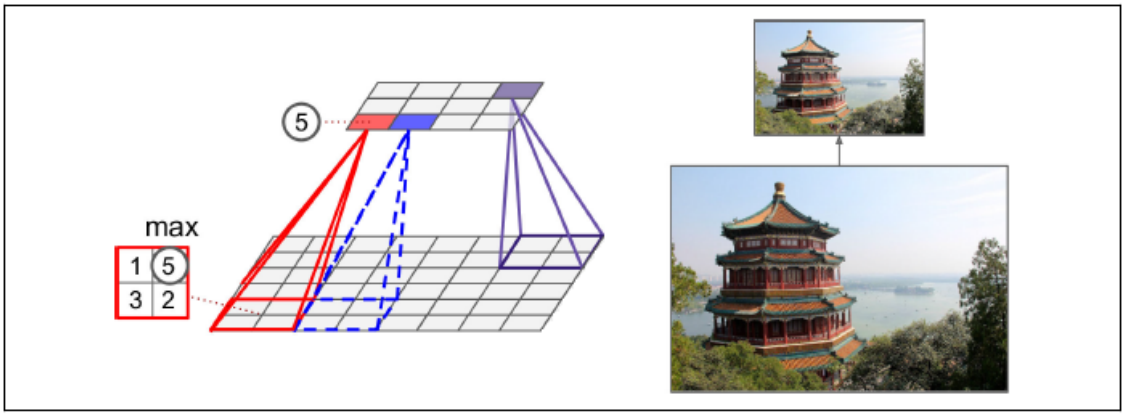


Figura 2.12. Capa de agrupamiento máximo (*max pooling layer*). Tamaño del núcleo 2×2, $S_W = S_H = 2$, $P = 0$.[18]

Así como las capas de agrupamiento son una técnica de optimización de rendimiento computacional para el caso de CNN, también es apropiado hacer uso de la transferencia de aprendizaje, otro método de optimización especialmente recomendable para conjuntos de entrenamiento pequeños.

2.4.4. Transferencia de aprendizaje

La solución a la mayoría de los problemas que involucran imágenes se plantea usando múltiples capas convolucionales, lo que es un ejemplo de RED NEURONAL PROFUNDA (DNN, por sus siglas en inglés *Deep Neural Network*). En los casos en los que no se cuente con suficiente cantidad de ejemplos de entrenamiento, no suele ser conveniente entrenar una DNN desde cero, es decir comenzar con valores de pesos aleatorios. Más bien lo recomendable en estas situaciones es hacer uso de alguna técnica de TRANSFERENCIA DE APRENDIZAJE (*transfer learning*) [80], como lo es el uso de modelos pre-entrenados.

Si bien esta técnica no es exclusivas de las CNN es ampliamente utilizadas por la comunidad, ya que grandes empresas como Google, Microsoft, Amazon, entre otras, han invertido muchísimo tiempo y dinero en el desarrollo de modelos que involucran el entrenamiento de grandes CNN en enormes conjuntos de datos para el reconocimiento de imágenes, como lo

[18]Imagen adaptada de [76].

son AlexNet, GoogLeNet, ResNet, InceptionNet, VGGNet, etc. Estos modelos entrenados han sido liberados, lo que permite a investigadores y profesionales que no cuentan con los recursos, datos o tiempo suficiente para entrenar redes de esta magnitud, usar dichos modelos como base en tareas de aprendizaje supervisado.

La idea básica del aprendizaje por transferencia es que los filtros aprendidos por las capas convolucionales de estas grandes redes pueden ser usados para la mayoría de las tareas basadas en el reconocimiento de imágenes, no sólo para las que fueron entrenadas originalmente. Así, se toma algún modelo ya aprendido, se quitan las últimas capas ocultas convolucionales (que son las encargadas de reconocer las características finales para las que fue entrenado dicho modelo) así como la capa de salida, se "congelan" los filtros aprendidos en el resto de las capas, ya que son valores que no necesitan ser actualizados, y se apilan nuevas capas convolucionales y de salida, que serán las encargadas de reconocer las características propias para la nueva tarea.

Hacer uso del aprendizaje por transferencia es sumamente recomendable, ya que no sólo ahorra tiempo de entrenamiento, sino que es especialmente bueno cuando no se dispone de grandes bases de datos.

Las nociones de bloques residuales y conexiones de atajo que vimos para MLP siguen siendo válidas para CNN, salvo que ahora la suma es entre las componentes de los mapas de características.

2.4.5. Aprendizaje residual en CNN

Como se explicó en la Sección 2.3.7, se considera un bloque residual a $\mathbf{y} = F_{\mathbf{w}}(\mathbf{x}) + \mathbf{x}$, donde en este caso $F_{\mathbf{w}}(\mathbf{x})$ representa la función residual a aprender luego de múltiples capas convolucionales; y la suma entre dos mapas de características se lleva a cabo lugar a lugar [39].

Existen varias situaciones en las que el tensor de entrada no tiene el mismo tamaño del tensor de salida. Para ejemplificar cómo realizar la suma correspondiente tomemos el caso en que los tensores sólo difieren en la última dimensión, es decir, difiere el número de filtros, pero no el tamaño del mini lote ni el tamaño de las imágenes (por ejemplo, basta tomar relleno de ceros $P = (f_H - 1)/2$ para imágenes cuadradas, con $f_H = f_W$ y paso $S = 1$ para asegurar esto último). Así, si $\mathbf{x}$ es de tamaño $|B_s| \times W \times H \times D_1$ y $F_{\mathbf{w}}(\mathbf{x})$ es de tamaño $|B_s| \times W \times H \times D_2$, donde $D_2 > D_1$, entonces se procede a rellenar con $P = D_2 - D_1$ ceros en la última dimensión para igualar así la forma de los tensores.

Finalmente, concluimos esta Sección nombrando algunas tareas que se resuelven usando CNN.

2.4.6. Uso de las CNN

Las CNN se hallan presentes en la resolución de diversos problemas de percepción visual, como son los servicios de búsqueda de imágenes, los vehículos autónomos y los sistemas de clasificación automática de videos, entre otros. Pero además, no se limitan a esto, sino que también tienen éxito en muchas otras tareas como el reconocimiento de voz o el procesamiento del lenguaje natural. Dado que trabajamos con imágenes, nos centraremos ahora en algunas de las aplicaciones visuales.

Se diferencian la clasificación de objetos, la detección de objetos y la segmentación de objetos en imágenes [83, 65], como se muestra en la Figura 2.13. El primer caso es el ya mencionado en la Sección 2.2.2, donde la clasificación podría ser multi-etiqueta o multi-clase, dependiendo de la cantidad y tipo de objetos presente en las imágenes. Por otro lado, localizar un único objeto en una imagen puede expresarse como un problema de clasificación y regresión, donde se predicen las coordenadas de los vértices superior izquierdo e inferior derecho de una caja o rectángulo que delimita al objeto (*bounding box*), o las coordenadas del centro del objeto junto con su altura y anchura; mientras que localizar y clasificar múltiples objetos es lo que se conoce como detección de objetos en imágenes.

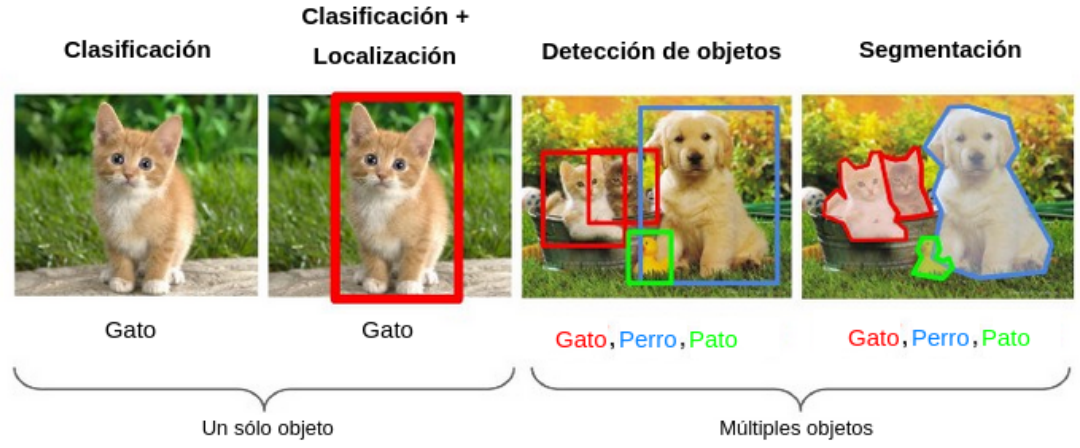


Figura 2.13. Comparación entre clasificación, detección y segmentación de objetos en imágenes.[19]

Por otro lado, si bien aplicar modelos de detección de objetos, nos permite construir un cuadro delimitador correspondiente a cada clase de la imagen, no sabemos nada sobre la forma del objeto, ya que los cuadros delimitadores tienen forma rectangular o cuadrada. Para esto se usa la segmentación de objetos en imágenes, que consiste en usar modelos que crean una máscara de píxeles para cada objeto de la imagen.

[19]Imagen adaptada de [65].

2.5. Detección de objetos

El problema de CLASIFICAR Y LOCALIZAR múltiples objetos en una imagen es lo que se conoce como DETECCIÓN DE OBJETOS. Existen métodos de detección de una etapa, que priorizan la velocidad de predicción, y métodos de dos etapas, que priorizan la precisión de la detección. Los modelos de una etapa incluyen YOLO [41], SSD [40] y RetinaNet [52], mientras que los de dos etapas incluyen Faster R-CNN [42], Mask R-CNN [48] y Cascade R-CNN [74], entre otros. Una comparación de éstos y otros métodos, así como un resumen de lo que implica la detección de objetos en imágenes se puede encontrar en [87, 88, 96, 65]. En este trabajo usamos la arquitectura de red de YOLO [41, 67], la cual explicaremos en detalle en la Sección 2.5.3.

El conjunto de entrenamiento para estos problemas está conformado por imágenes junto con archivos que guarden la información del rectángulo delimitador y de la clase a la cual pertenece cada objeto de interés presente en la imagen (por ejemplo, si se trata de un perro o un gato). Por otro lado, la salida de los algoritmos de detección consiste en las coordenadas predichas de los vértices superior izquierdo e inferior derecho del rectángulo delimitador (o las coordenadas de su centro, y altura y anchura), junto con el nombre de la clase del objeto y su nivel de certeza o confianza. Solo se muestran las salidas cuyo nivel de certeza supere cierto valor umbral, el cual es fijado con un hiperparámetro. Para evaluar si un modelo detectó correctamente, se tienen en cuenta los conceptos de la próxima Sección.

2.5.1. Medidas de rendimiento

En [97] vemos que si bien la salida de los detectores está compuesta por una lista de cuadros delimitadores, niveles de confianza y clases, el formato estándar del archivo de salida varía mucho según los distintos algoritmos de detección. Las detecciones de cuadros delimitadores se representan en su mayoría por las coordenadas de los vértices superior izquierdo e inferior derecho $(x_{min}, y_{min}, x_{max}, y_{max})$, mientras que YOLO establece los cuadros delimitadores por las coordenadas de su centro, su anchura y altura $(\frac{x_{centro}}{\text{ancho imagen}}, \frac{y_{centro}}{\text{alto imagen}}, \frac{\text{ancho cuadro}}{\text{ancho imagen}}, \frac{\text{alto cuadro}}{\text{alto imagen}})$.

Existen diferentes retos y competencias de detección de objetos que evalúan el rendimiento de la detección. En estos eventos, los participantes reciben un conjunto de imágenes no etiquetadas como conjunto de testeo, en las que los objetos tienen que ser detectados utilizando los modelos propuestos por los participantes. Algunos concursos proporcionan su propio código fuente (o el de terceros), lo que permite a los participantes evaluar sus algoritmos en un conjunto de imágenes etiquetadas de validación antes de presentar sus detecciones en el conjunto de testeo. En la mayoría de las competencias se utiliza la precisión media (AP), la cual explicaremos a continuación una vez comprendido el concepto de Intersección sobre Unión, para poder definir la curva precisión-exhaustividad.

Intersección sobre unión

La INTERSECCIÓN SOBRE UNIÓN IoU (*Intersection over Union*) es la medida de rendimiento utilizada en los problemas de detección, la cual calcula el área de intersección entre la caja delimitadora etiquetada y la predicha por el modelo, y lo divide por el área de la unión de ambos rectángulos. Si llamamos B_{GT} al rectángulo delimitador etiquetado (del inglés, *ground truth bounding box*) y B_P al rectángulo delimitador predicho (del inglés, *predicted bounding box*), entonces:

$$\text{IoU} = \frac{B_{GT} \cap B_P}{B_{GT} \cup B_P},$$

como se muestra en la Figura 2.14.

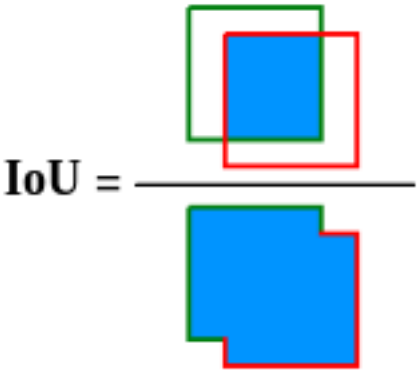


Figura 2.14. Cálculo de intersección sobre unión.[20]

Al fijar con un hiperparámetro un valor umbral de IoU (usualmente 0,5), se establece cuándo una predicción será VP, FP o FN. En los problemas de detección no se evalúan los VN, ya que existen infinitas cajas delimitadoras que no deben ser detectadas en cada imagen.

Un rectángulo B_P se considera VP si el valor de IoU con el rectángulo B_{GT} supera el umbral. Por otro lado, se considera FP si el valor de IoU es menor al umbral (incluyendo IoU = 0, que es cuando no hay intersección entre los rectángulos), o cuando hay más de un B_P para un B_{GT}, donde el de mayor nivel de certeza será VP y los demás serán considerados FP. Por último, se considera FN si un B_{GT} no es detectado, o si existe un B_P para un B_{GT} con un IoU mayor al umbral, pero que la clase predicha sea incorrecta (por ejemplo, el modelo detecta dónde está ubicado un gato, pero le asigna la clase "perro"). Entonces, teniendo en cuenta lo anterior, podemos escribir la precisión (*precision*) y exhaustividad (*recall*) como:

$$\textit{precision} = \frac{\text{VP}}{\text{todas las detecciones}} \qquad \textit{recall} = \frac{\text{VP}}{\text{todas las etiquetas}}$$

[20]Imagen adaptada de [97].

Curva Precisión-Exhaustividad

La CURVA PRECISIÓN-EXHAUSTIVIDAD muestra el compromiso entre los valores de precisión y exhaustividad para diferentes niveles de certeza asociados a los rectángulos predichos. Un buen detector de objetos debería encontrar todos los rectángulos etiquetados (es decir, FN $= 0 \equiv$ exhaustividad alta) mientras que sólo identifica los objetos relevantes (es decir, FP $= 0 \equiv$ precisión alta). Por ende, un buen detector es aquel cuya precisión se mantiene alta a medida que aumenta su exhaustividad. Una forma de medir esto es calculando el ÁREA BAJO LA CURVA (AUC, por sus siglas en inglés *Area Under the Curve*): a mayor AUC mejor el detector.

La curva precisión-exhaustividad suele ser una curva en zigzag, lo que representa una dificultad para una medición precisa de su AUC con métodos de integración numérica. Una solución a esta problemática es procesar la curva para eliminar el comportamiento en zigzag antes de la estimación del AUC, por medio de la interpolación, el cual es un método que estima el valor que toma una función a partir de datos conocidos, por lo que suele usarse para aproximar funciones: dada una función $f : [a, b] \to \mathbb{R}$ y un conjunto de puntos del dominio $x_1, x_2, \ldots, x_n \in [a, b]$, existe una función $p : [a, b] \to \mathbb{R}$ tal que $f(x_i) = p(x_i)$ para toda $i = 1, 2, \ldots, n$, y se dice que p INTERPOLA a f en esos puntos. En general una interpolación no necesariamente es una buena aproximación, pero existen condiciones conocidas en las que sí. Para profundizar en el tema se sugiere consultar [9].

Entonces, para aproximar la curva precisión-exhaustividad se pueden utilizar dos enfoques: la interpolación de 11 puntos y la interpolación de todos los puntos. El área calculada así se denomina precisión promedio y se denota AP por sus siglas en inglés (*Average Precision*). En la interpolación de 11 puntos, se aproxima la curva usando los valores de precisión P máximos para un conjunto de 11 valores de exhaustividad equidistantes $R = 0;\ 0{,}1;\ 0{,}2; \ldots; 1$, esto es:

$$\text{AP}_{11} = \frac{1}{11} \sum_{R \in \{0,\ 0{,}1,\ldots,0{,}9,1\}} P_{\text{interp}}(R), \qquad \text{donde} \qquad P_{\text{interp}}(R) = \max_{\tilde{R}:\tilde{R} \geq R} P(\tilde{R})$$

En esta definición de AP, en lugar de usar la precisión $P(R)$ observada en cada nivel de exhaustividad R, la AP se obtiene considerando $P_{\text{interp}}(R)$, que es la máxima precisión para valores de exhaustividad mayores o iguales a R.

Por otro lado, en la interpolación de todos los puntos, como su nombre lo indica, no se usan sólo 11, sino todos los puntos:

$$\text{AP}_{\text{todos}} = \sum_n (R_{n+1} - R_n) P_{\text{interp}}(R_{n+1}), \qquad \text{donde} \qquad P_{\text{interp}}(R_{n+1}) = \max_{\tilde{R}:\tilde{R} \geq R_{n+1}} P(\tilde{R})$$

En este caso, en lugar de utilizar la precisión observada en sólo unos pocos puntos, la AP se obtiene interpolando la precisión en cada nivel de exhaustividad, $P_{\text{interp}}(R_{n+1})$, tomando la máxima precisión para valores de exhaustividad mayores o iguales a R_{n+1}.

La media AP (que se denota mAP, por sus siglas en inglés *mean Average Precision*) es una medida de rendimiento utilizada para medir la precisión de un detector de objetos sobre

todas las clases de una base de datos determinada. El mAP es simplemente el promedio de AP sobre todas las clases, es decir,

$$\text{mAP} = \frac{1}{C}\sum_{i=1}^{C}\text{AP}_i,$$

donde AP_i es la AP de la $i-$ésima clase y C es el total de clases presentes en la base de datos.

El cálculo de las medidas de rendimiento depende de la competencia o contexto donde serán evaluadas las detecciones. A continuación mencionamos las más importantes, junto con las medidas adoptadas por cada una.

Algunas competencias

En la mayoría de las competencias se utiliza la precisión promedio (AP) o alguna de sus variantes, como medida de rendimiento para evaluar a los competidores. Por ejemplo, el desafío PASCAL VOC [20] proporciona el código fuente para medir la AP y su media (mAP) sobre todas las clases de objetos (20 en total) como explicamos en la Sección anterior, usando un valor umbral de IoU de 0,5 y, a partir del 2010, la interpolación de todos los puntos (antes de eso se usaba la interpolación de 11 puntos). También los retos *Open Images* [82] y *Google AI Open Images* [68] utilizan mAP, mientras que la competencia *ImageNet Object Localization* [94] no recomienda ningún código para calcular su métrica de evaluación, pero proporciona un pseudocódigo que lo explica.

Por otro lado, la competencia de detección COCO (de rectángulos delimitadores) [109] utiliza varias medidas de rendimiento, que se diferencian de la de PASCAL VOC [97, 60]:

- AP: para COCO, AP es la precisión promedio sobre múltiples valores de IoU (es decir, para diferentes valores umbrales de IoU que determinan un VP). La notación AP@[.50:.05:.95] indica que se usan diez valores de IoU desde 0,5 a 0,95 con un paso de 0,05. También pueden evaluarse para un solo valor de IoU, donde los más usuales son 0,5 (que es la medida usada en PASCAL VOC) y 0,75, denotados como AP@.50 y AP@.75, respectivamente.

- AP Diferentes Escalas: se determina la AP para objetos de diferentes tamaños: pequeño (área $< 32^2$ píxeles), mediano ($32^2 <$ área $< 96^2$ píxeles), y grande (área $> 96^2$ píxeles).

- Exhaustividad Promedio (AR, por sus siglas en inglés *Average Recall*): se estima usando los valores máximos de exhaustividad dado un número fijo de detecciones por imagen (1, 10 ó 100), promediando para los diez valores de IoU usados para calcular AP y todas las clases (80 en total).

- AR Diferentes Escalas: se determina la AR para los objetos en los mismos tamaños usados en la AP Diferentes Escalas.

Más aún, para COCO no hay distinción entre mAP y AP [110]: "AP es promediado sobre todas las categorías. Tradicionalmente, esto es llamado 'media de la precisión promedio' (mAP). No hacemos distinción entre AP y mAP (así como entre AR y mAR) y asumimos que la diferencia está clara por contexto".

En este trabajo reportaremos la medida de rendimiento mAP calculada como en las competencias PASCAL VOC.

Ejemplo de cálculo de AP

Para mostrar cómo se calcula la precisión promedio veremos un ejemplo extraído de [97]. Supongamos que tenemos solamente una clase de objetos. En la Figura 2.15 los rectángulos verdes representan los objetos reales a detectar, mientras que los rectángulos rojos (identificados por letras mayúsculas, de la A a la Y) representan las detecciones obtenidas por el modelo, y los porcentajes asociados a las letras representan los niveles de certeza. Para evaluar la precisión y exhaustividad de las 24 detecciones correspondientes a los 15 objetos etiquetados distribuidos en siete imágenes, establecemos el umbral de IoU en 0,3 para determinar si una detección es considerada VP o FP.

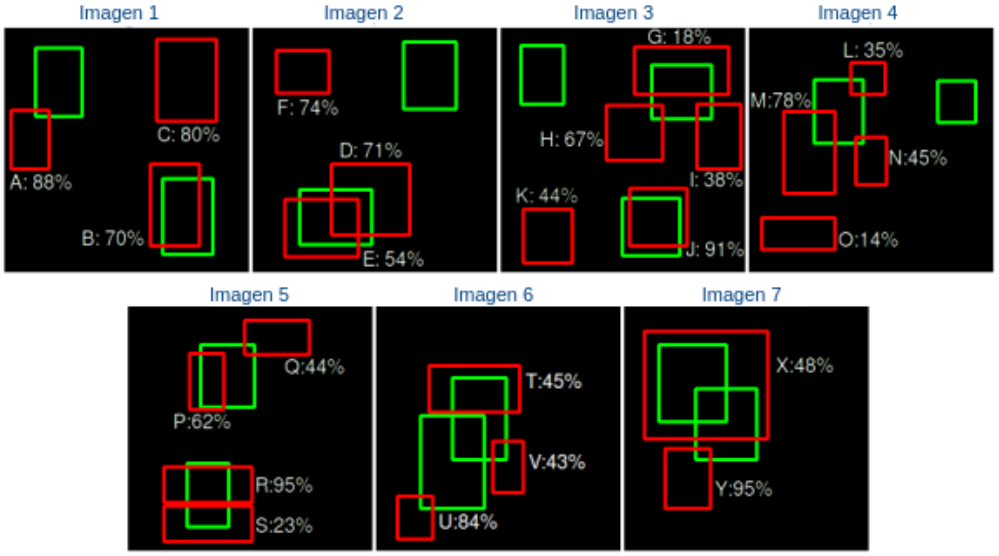


Figura 2.15. 24 detecciones (rojo) realizadas por un detector de objetos cuyo objetivo era detectar 15 objetos reales (verde), de una misma clase.[21]

El Cuadro 2.1 presenta cada detección ordenada según su nivel de certeza. Si el valor IoU entre la detección y el rectángulo etiquetado es mayor al 30 %, entonces se identifica con un 1 la columna **VP**, caso contrario se usa un 0 y se considera 1 en **FP**. En casos donde existe más de una detección intersectando a un rectángulo etiquetado (como las detecciones D y E en la imagen 2, ó G, H, I en la imagen 3), de a cuerdo a lo establecido en

[21] Imagen adaptada de [97].

la compentencia PASCAL VOC, se considera VP la detección de mayor nivel de certeza, mientras que el resto es considerada FP. Las columnas **VP acum** y **FP acum** acumulan la cantidad total de VP y FP a lo largo de todas las detecciones por encima de la que se está analizando. El valor de la columna **precisión** se calcula en cada fila como

$$\frac{\text{VP acum}}{\text{VP acum} + \text{FP acum}},$$

mientras que la **exhaustividad** se calcula en cada fila como

$$\frac{\text{VP acum}}{\text{total detecciones}} = \frac{\text{VP acum}}{24}.$$

Cuadro 2.1. Cálculo de precisión y exhaustividad para umbral de IoU de 30 %.

detección	certeza	VP	FP	VP acum	FP acum	precisión	exhaustividad
R	95 %	1	0	1	0	1	0,0666
Y	95 %	0	1	1	1	0,5	0,0666
J	91 %	1	0	2	1	0,6666	0,1333
A	88 %	0	1	2	2	0,5	0,1333
U	84 %	0	1	2	3	0,4	0,1333
C	80 %	0	1	2	4	0,3333	0,1333
M	78 %	0	1	2	5	0,2857	0,1333
F	74 %	0	1	2	6	0,25	0,1333
D	71 %	0	1	2	7	0,2222	0,1333
B	70 %	1	0	3	7	0,3	0,2
H	67 %	0	1	3	8	0,2727	0,2
P	62 %	1	0	4	8	0,3333	0,2666
E	54 %	1	0	5	8	0,3846	0,3333
X	48 %	1	0	6	8	0,4285	0,4
N	45 %	0	1	6	9	0,4	0,4
T	45 %	0	1	6	10	0,375	0,4
K	44 %	0	1	6	11	0,3529	0,4
Q	44 %	0	1	6	12	0,3333	0,4
V	43 %	0	1	6	13	0,3157	0,4
I	38 %	0	1	6	14	0,3	0,4
L	35 %	0	1	6	15	0,2857	0,4
S	23 %	0	1	6	16	0,2727	0,4
G	18 %	1	0	7	16	0,3043	0,4666
O	14 %	0	1	7	17	0,2916	0,4666

Entonces, para estos valores realizamos la Curva Precisión-Exhaustividad, la cual se muestra en la Figura 2.16.

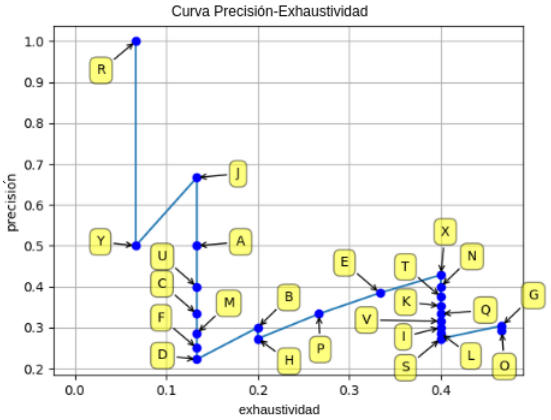


Figura 2.16. Curva Precisión-Exhaustividad, cuyos valores fueron calculados para cada detección en el Cuadro 2.1.[22]

Cada método de interpolación (ilustrados en la Figura 2.17) lleva a un valor de AP diferente. Por un lado, para la interpolación de 11 puntos tenemos:

$$
\begin{aligned}
P_{\text{interp}}(0) &= \max_{\tilde{R}:\tilde{R}\geq 0} P(\tilde{R}) = 1 \\
P_{\text{interp}}(0{,}1) &= \max_{\tilde{R}:\tilde{R}\geq 0{,}1} P(\tilde{R}) = 0{,}6666 \\
P_{\text{interp}}(0{,}2) &= \max_{\tilde{R}:\tilde{R}\geq 0{,}2} P(\tilde{R}) = 0{,}4285 \\
P_{\text{interp}}(0{,}3) &= \max_{\tilde{R}:\tilde{R}\geq 0{,}3} P(\tilde{R}) = 0{,}4285 \\
P_{\text{interp}}(0{,}4) &= \max_{\tilde{R}:\tilde{R}\geq 0{,}4} P(\tilde{R}) = 0{,}4285 \\
P_{\text{interp}}(0{,}5) &= P_{\text{interp}}(0{,}6) = P_{\text{interp}}(0{,}7) = P_{\text{interp}}(0{,}8) = P_{\text{interp}}(0{,}9) = P_{\text{interp}}(1) = 0
\end{aligned}
$$

Entonces,

$$\text{AP}_{11} = \frac{1}{11}(1 + 0{,}6666 + 0{,}4285 + 0{,}4285 + 0{,}4285) = 0{,}2684$$

Por otro lado, para la interpolación de todos los puntos tenemos:

$$
\begin{aligned}
\text{AP}_{\text{todos}} &= 1 \times (0{,}0666 - 0) \; + \; 0{,}6666 \times (0{,}1333 - 0{,}0666) \\
&\quad + \; 0{,}4285 \times (0{,}4 - 0{,}1333) \; + \; 0{,}3043 \times (0{,}4666 - 0{,}4) \\
&= 0{,}2456
\end{aligned}
$$

Finalmente, ya sea que elijamos la interpolación de 11 puntos o la de todos los puntos, como en este caso sólo tenemos una sola clase, el valor mAP coincide con la precisión promedio.

[22]Imagen adaptada de [97].

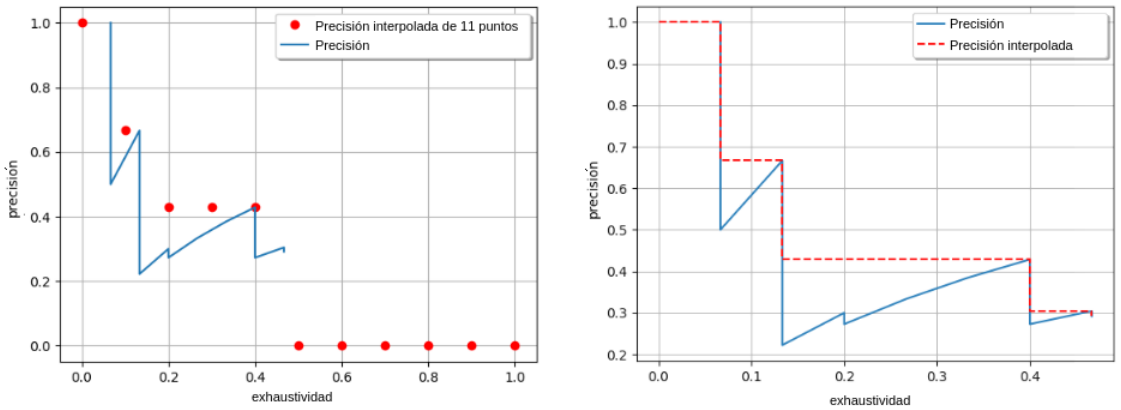


Figura 2.17. Interpolación de 11 puntos (izquierda) e interpolación de todos los puntos (derecha).[23]

Uno de los problemas presentes en la detección de objetos en imágenes es que el algoritmo puede encontrar múltiples detecciones del mismo objeto.

2.5.2. Supresión de no máximos

Para solucionar el problema de las múltiples detecciones de un mismo objeto se utiliza la SUPRESIÓN DE NO MÁXIMOS (NMS, por sus siglas en inglés *non-maximum suppression*), que es un método que selecciona solo una detección de entre muchas superpuestas.

Para utilizar este método se fijan dos hiperparámetros: uno de ellos es el umbral en el nivel de confianza de una detección, el cual se denota OBJ, y el otro es un valor umbral de IoU que compara entre las múltiples detecciones, el cual se llama NMS. Al aplicar este método, en primer lugar se descartan todas las detecciones con un nivel de certeza menor al valor umbral OBJ. Luego se elige la detección cuyo nivel de confianza sea máximo, y se la compara contra el resto de las detecciones: aquellas cuya IoU sea mayor o igual al valor umbral NMS se suprimen. Luego se selecciona la siguiente detección con mayor nivel de confianza y se repite el procedimiento. Notemos que, a diferencia del valor umbral de IoU explicado en la Sección 2.5.1, que es usado para comparar rectángulos detectados con rectángulos etiquetados, el NMS es el valor umbral del cálculo de IoU que se emplea entre detecciones (sin comparar contra los rectángulos etiquetados).

Para terminar este capítulo nos adentraremos en la estructura de red neuronal YOLO, la cual hemos mencionado antes, ya que será la encargada de generar modelos de detección a partir de las imágenes de nuestros conjuntos de entrenamiento, los cuales serán descriptos en el Capítulo 3 (Sección 3.2). Para esto veremos el tipo y distribución de las capas que la componen, así como las llamadas cajas de anclaje que utiliza para detectar, y cómo es la

[23]Imagen adaptada de [97].

función error a minimizar durante la propagación hacia atrás.

2.5.3. YOLO

YOLO fue propuesto en 2015 por Redmon, Divvala, Girshick y Farhadi en [41], convirtiéndose en el primer detector de una etapa, y luego fue mejorado por Redmon y Farhadi en [53, 67]. Se caracteriza por ser muy rápido, por ejemplo la primera versión corre a 45 fps (cuadros o fotogramas por segundo, del inglés *frames per second*), mientras que una versión rápida de la misma logra 150 fps, lo que lo hace un sistema de detección de objetos en imágenes y videos en tiempo real. Las modificaciones propuestas YOLOv2 [53] y YOLOv3 [67] son más rápidas, además que mejoraron la estructura de la red y la manera de predecir las coordenadas de los rectángulos, así como modificaron en consecuencia la función error a minimizar. Su nombre proviene de las iniciales de *You Only Look Once*, que significa "sólo miras una vez". A diferencia de los clasificadores clásicos, que desplazan una "ventana" a lo largo y alto de la imagen buscando los objetos de interés, YOLO mira la imagen sólo una vez para detectar múltiples objetos.

Otros detectores como R-CNN y sus modificaciones Fast R-CNN y Faster R-CNN, por un lado proponen regiones de interés y generan candidatos para cuadros delimitadores, luego determinan los niveles de certeza de las predicciones y ajustan los rectángulos predichos, y por otro lado aplican supresión de no máximos. Cada etapa de este complejo proceso debe ajustarse con precisión de forma independiente y el sistema resultante es muy lento, por ejemplo R-CNN tarda más de 40 segundos por imagen en la etapa de testeo. Si bien YOLO comparte algunas similitudes con éstos, su sistema combina todos estos componentes individuales en un único modelo optimizado conjuntamente, lo que lo hace incluso más rápido que las versiones Fast y Faster de R-CNN [41]. Es por esto que elegimos YOLO como detector, ya que si bien planteamos inicialmente el trabajo con imágenes, no descartamos la posibilidad de detectar en videos, siendo YOLO la mejor opción en términos de velocidad.

En Febrero 2020 Redmon anunció [98] que no seguiría realizando investigaciones en el área de *Computer Vision* dado que al ver el impacto que tenía su trabajo le preocuparon las posibles aplicaciones militares, así como los problemas de privacidad, como manifestó también al final de [67]. Sin embargo, Bochkovskiy, Wang y Liao presentaron la cuarta versión YOLOv4 en [90] y [100]. Más aún, otros autores presentaron una quinta versión, que aunque no cuenta con un paper formal ya lleva cinco actualizaciones, siendo la más reciente en Enero de este año [103]. Más allá de esto, nosotros usamos YOLOv3 ya que estas nuevas versiones fueron publicadas con posterioridad a cuando comenzamos este trabajo.

Estructura de YOLOv3

La estructura de YOLOv3 está conformada por dos grandes bloques que usan capas convolucionales: por un lado se tiene un extractor de características, y por otro el detector

que proporciona las predicciones, basándose en lo extraído en el primer bloque. YOLO predice las coordenadas del centro de cada rectángulo que contiene un objeto de interés, junto con su altura y anchura, el nivel de certeza de la detección, y la clase a la cual pertenece el objeto.

El extractor de características de la tercera versión de este sistema (YOLOv3) [67], se llama *Darknet-53*, el cual está compuesto por 52 capas convolucionales con $P = 1$, cada una seguida por una capa BatchNorm y función de activación Leaky ReLU, sin capas de agrupamiento, como se ve en detalle en [66]. Una representación esquemática de *Darknet-53* se muestra en la Figura 2.18, mientras que la arquitectura general se muestra en el Cuadro 2.2: cada rectángulo es un bloque residual (por eso la presencia de conexiones de atajo), cuyas capas convolucionales tienen $S = 1$, pero además tiene capas convolucionales entre los bloques con $S = 2$ que reducen la dimensión de los mapas de características.

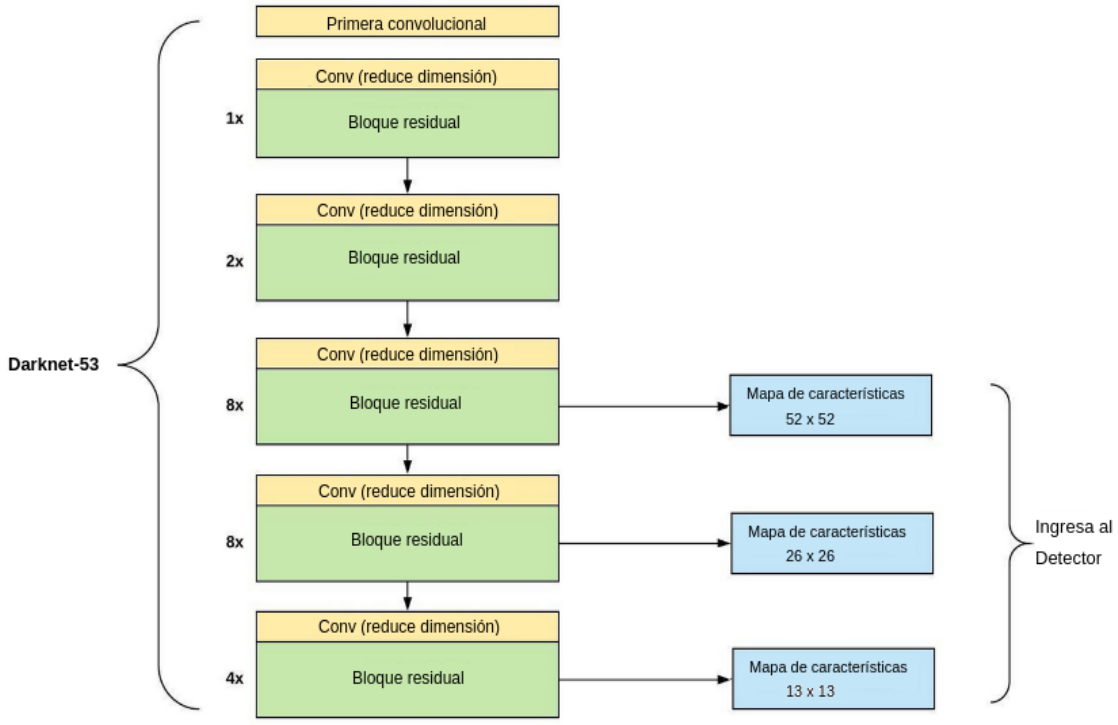


Figura 2.18. Estructura del extractor de características *Darknet-53*.[24]

Los tamaños de las entradas y salidas de las capas descriptas surgen de asumir que las imágenes de la base de datos son a color y de tamaño 416×416. Por último, cabe destacar que YOLOv3 también puede ser utilizado en problemas que sólo incluyen tareas de clasificación, para dichos casos se agrega al final de *Darknet-53* (es decir, luego de la capa 75) una capa de agrupamiento promedio y una capa densa [67, 79].

Para las tareas de detección, en lugar de las capas de agrupamiento promedio y densa, se agregan 32 capas (detalladas a partir de la línea 549 de [66]), entre las que se incluyen capas

[24]Imagen adaptada de [79].

Cuadro 2.2. Estructura de *Darknet-53*.[25]

Capa		Tipo	Filtros	Tamaño	Entrada	Salida
1		Convolucional	32	$3 \times 3 \,/\, 1$	$416 \times 416 \times 3$	$416 \times 416 \times 32$
2		Convolucional	64	$3 \times 3 \,/\, 2$	$416 \times 416 \times 32$	$208 \times 208 \times 64$
3		Convolucional	32	$1 \times 1 \,/\, 1$	$208 \times 208 \times 64$	$208 \times 208 \times 32$
4	$1\times$	Convolucional	64	$3 \times 3 \,/\, 1$	$208 \times 208 \times 32$	$208 \times 208 \times 64$
5		Atajo			$208 \times 208 \times 64$	$208 \times 208 \times 64$
6		Convolucional	128	$3 \times 3 \,/\, 2$	$208 \times 208 \times 64$	$104 \times 104 \times 128$
		Convolucional	64	$1 \times 1 \,/\, 1$	$104 \times 104 \times 128$	$104 \times 104 \times 64$
$7 - 12$	$2\times$	Convolucional	128	$3 \times 3 \,/\, 1$	$104 \times 104 \times 64$	$104 \times 104 \times 128$
		Atajo			$104 \times 104 \times 128$	$104 \times 104 \times 128$
13		Convolucional	256	$3 \times 3 \,/\, 2$	$104 \times 104 \times 128$	$52 \times 52 \times 256$
		Convolucional	128	$1 \times 1 \,/\, 1$	$52 \times 52 \times 256$	$52 \times 52 \times 128$
$14 - 37$	$8\times$	Convolucional	256	$3 \times 3 \,/\, 1$	$52 \times 52 \times 128$	$52 \times 52 \times 256$
		Atajo			$52 \times 52 \times 256$	$52 \times 52 \times 256$
38		Convolucional	512	$3 \times 3 \,/\, 2$	$52 \times 52 \times 256$	$26 \times 26 \times 512$
		Convolucional	256	$1 \times 1 \,/\, 1$	$26 \times 26 \times 512$	$26 \times 26 \times 256$
$39 - 62$	$8\times$	Convolucional	512	$3 \times 3 \,/\, 1$	$26 \times 26 \times 256$	$26 \times 26 \times 512$
		Atajo			$26 \times 26 \times 512$	$26 \times 26 \times 512$
63		Convolucional	1024	$3 \times 3 \,/\, 2$	$26 \times 26 \times 512$	$13 \times 13 \times 1024$
		Convolucional	512	$1 \times 1 \,/\, 1$	$13 \times 13 \times 1024$	$13 \times 13 \times 512$
$64 - 75$	$4\times$	Convolucional	1024	$3 \times 3 \,/\, 1$	$13 \times 13 \times 512$	$13 \times 13 \times 1024$
		Atajo			$13 \times 13 \times 1024$	$13 \times 13 \times 1024$

convolucionales y conexiones de atajo como las de *Darknet-53*, pero también de nuevos tipos: capas *upsample* que aumentan el tamaño de la imagen usando interpolación bilineal; capas *route* que tienen por salida el mapa de características de una capa en particular, o que realizan una concatenación de los mapas de características de dos capas, que consiste en aumentar la dimensión de profundidad uniendo las profundidades de ambas capas; y capas *yolo* que son las encargadas de la detección, las cuales son precedidas por convolucionales con núcleos 1×1 con función de activación lineal.

YOLOv3 se caracteriza por ser un detector de escala múltiple, es decir que detecta objetos de pequeña, mediana y gran escala, para lo cual es necesario la extracción de características en múltiples escalas. Es por esto que se aprovechan las características provenientes de los tres últimos bloques residuales de *Darknet-53* en la etapa de detección. Luego de tomar el mapa de características de 13×13 (escala grande) del último bloque residual, se usan múltiples capas convolucionales de núcleos 1×1 y 3×3 de activación

[25]Cuadro adaptado de [67] y modificado según [69] y [66]. Si bien en `Python` se comienza a numerar desde 0, para este cuadro decidimos asignarle a la capa de entrada, es decir la primera capa, el número 1.

Leaky ReLU antes de una convolucional final de núcleo 1×1 de activación lineal. Para las escalas mediana y chica se concatenan mapas de características de capas anteriores, así la escala de detección mediana se benefica de los resutados de la escala grande, y la pequeña se beneficia de las escalas mediana y grande [67, 79, 66, 69].

Asumiendo que las imágenes son a color de tamaño 416×416, y siguiendo a [66, 69], la primera detección (escala grande) se obtiene en la capa 83, luego de que la red redujo la dimensión de la imagen en un factor de 32, obteniéndose un mapa de características de $13 \times 13 \times (B \cdot (5 + C))$, donde B es la cantidad de rectángulos que una celda del mapa de características puede predecir, 5 es por las coordenadas del centro, ancho y alto del rectángulo, y el nivel de certeza (llamado *objectness*, sobre el cual profundizaremos en las próximas Secciones), y C es la cantidad de clases. Luego, se toma el mapa de características de la capa 80, y se pasa por una nueva capa convolucional antes de aumentar el tamaño de la imagen por $2\times$ a una dimensión de 26×26. Este mapa de características es concatenado en profundidad con el mapa de la capa 62. Luego se pasa por varias capas convolucionales y se obtiene la segunda detección (escala mediana) en la capa 95, dando un mapa de características de $26 \times 26 \times (B \cdot (5 + C))$. Con un procedimiento similar, se obtiene la tercera detección (escala pequeña) en la capa 107, la cual se obtiene luego de tomar el mapa de características de la capa 92, aplicarle una convolución, aumentar el tamaño de la imagen, concatenar los mapas de las capas 98 y 37, y pasar por nuevas capas convolucionales, dejando entonces un mapa de características de $52 \times 52 \times (B \cdot (5 + C))$. En el Cuadro 2.3 se muestra en detalle esta estructura encargada de la detección (es una continuación del Cuadro 2.2), así como una representación esquemática de la misma en la Figura 2.19.

Cuadro 2.3. Estructura de YOLOv3 encargada de la detección.[26]

Capa	Tipo	Filtros	Tamaño	Entrada	Salida
76	Convolucional	512	$1 \times 1 \,/\, 1$	$13 \times 13 \times 1024$	$13 \times 13 \times 512$
77	Convolucional	1024	$3 \times 3 \,/\, 1$	$13 \times 13 \times 512$	$13 \times 13 \times 1024$
78	Convolucional	512	$1 \times 1 \,/\, 1$	$13 \times 13 \times 1024$	$13 \times 13 \times 512$
79	Convolucional	1024	$3 \times 3 \,/\, 1$	$13 \times 13 \times 512$	$13 \times 13 \times 1024$
80	Convolucional	512	$1 \times 1 \,/\, 1$	$13 \times 13 \times 1024$	$13 \times 13 \times 512$
81	Convolucional	1024	$3 \times 3 \,/\, 1$	$13 \times 13 \times 512$	$13 \times 13 \times 1024$
82	Convolucional	$B \cdot (5 + C)$	$1 \times 1 \,/\, 1$	$13 \times 13 \times 1024$	$13 \times 13 \times (B \cdot (5 + C))$
83	YOLO 1				
84	route 80				
85	Convolucional	256	$1 \times 1 \,/\, 1$	$13 \times 13 \times 512$	$13 \times 13 \times 256$
86	upsample		$2\times$	$13 \times 13 \times 256$	$26 \times 26 \times 256$
87	route 86 62				
88	Convolucional	256	$1 \times 1 \,/\, 1$	$26 \times 26 \times 768$	$26 \times 26 \times 256$
89	Convolucional	512	$3 \times 3 \,/\, 1$	$26 \times 26 \times 256$	$26 \times 26 \times 512$
90	Convolucional	256	$1 \times 1 \,/\, 1$	$26 \times 26 \times 512$	$26 \times 26 \times 256$
91	Convolucional	512	$3 \times 3 \,/\, 1$	$26 \times 26 \times 256$	$26 \times 26 \times 512$
92	Convolucional	256	$1 \times 1 \,/\, 1$	$26 \times 26 \times 512$	$26 \times 26 \times 256$
93	Convolucional	512	$3 \times 3 \,/\, 1$	$26 \times 26 \times 256$	$26 \times 26 \times 512$
94	Convolucional	$B \cdot (5 + C)$	$1 \times 1 \,/\, 1$	$26 \times 26 \times 512$	$26 \times 26 \times (B \cdot (5 + C))$
95	YOLO 2				
96	route 92				
97	Convolucional	128	$1 \times 1 \,/\, 1$	$26 \times 26 \times 256$	$26 \times 26 \times 128$
98	upsample		$2\times$	$26 \times 26 \times 128$	$52 \times 52 \times 128$
99	route 98 37				
100	Convolucional	128	$1 \times 1 \,/\, 1$	$52 \times 52 \times 384$	$52 \times 52 \times 128$
101	Convolucional	256	$3 \times 3 \,/\, 1$	$52 \times 52 \times 128$	$52 \times 52 \times 256$
102	Convolucional	128	$1 \times 1 \,/\, 1$	$52 \times 52 \times 256$	$52 \times 52 \times 128$
103	Convolucional	256	$3 \times 3 \,/\, 1$	$52 \times 52 \times 128$	$52 \times 52 \times 256$
104	Convolucional	128	$1 \times 1 \,/\, 1$	$52 \times 52 \times 256$	$52 \times 52 \times 128$
105	Convolucional	256	$3 \times 3 \,/\, 1$	$52 \times 52 \times 128$	$52 \times 52 \times 256$
106	Convolucional	$B \cdot (5 + C)$	$1 \times 1 \,/\, 1$	$52 \times 52 \times 256$	$52 \times 52 \times (B \cdot (5 + C))$
107	YOLO 3				

[26]Si bien este Cuadro no está detallado en [67], lo construimos a partir de [66] y [69].

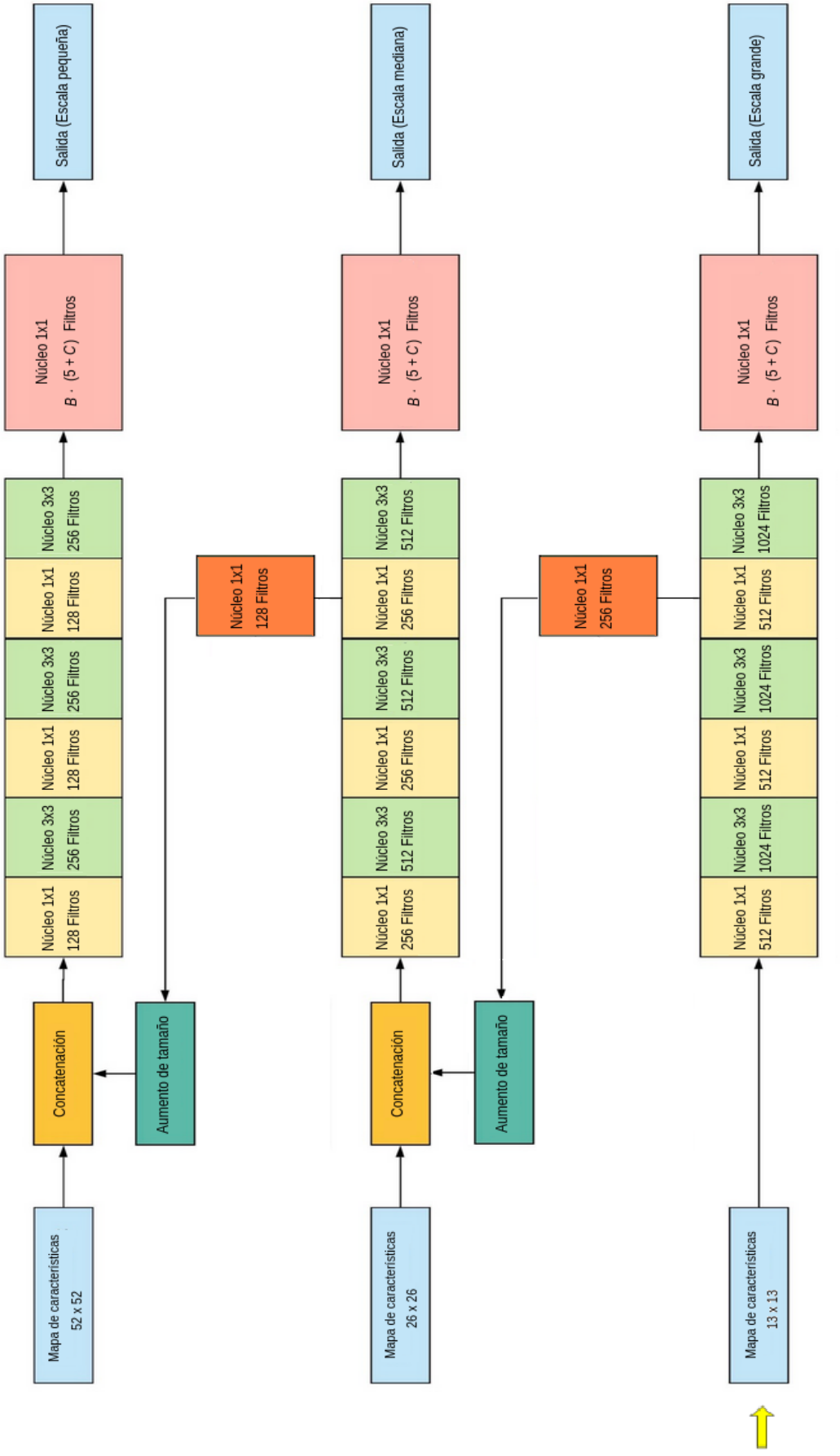


Figura 2.19. Detector de YOLOv3.[27]

En lugar de predecir las coordenadas absolutas de los rectángulos, YOLOv3 utiliza cajas de anclaje, las cuales explicamos a continuación.

Cajas de anclaje y predicciones

Como las predicciones se realizan usando capas convolucionales de núcleo 1×1, las salidas de las distintas escalas son un mapa de características. Así, si la imagen original se divide en un grillado de 13×13, 26×26 ó 52×52, según corresponda, se espera que una celda de dicho grillado prediga la ubicación de un objeto con un rectángulo delimitador, si el centro de ese objeto pertenece al campo receptivo efectivo (ERF) de esa celda [71].

Para facilitar la predicción de los rectángulos delimitadores (B_P), se utilizan las llamadas CAJAS DE ANCLAJE (*anchor boxes*), las cuales son rectángulos de tamaño prefijado, que son determinados antes del entrenamiento, ejecutando un algoritmo *K-Means* (tipo de aprendizaje no supervisado) sobre todo el conjunto de datos. Cada caja de anclaje tiene dimensiones alto y ancho diferentes. Por ejemplo, en la Figura 2.20 vemos posibles cajas de anclaje para las diferentes escalas en color celeste, siendo el recuadro amarillo el etiquetado y el rojo la celda que contiene el centro del objeto. En este caso particular podemos ver que el rectángulo etiquetado se corresponde con una de las cajas de escala grande (13×13).

Figura 2.20. Cajas de anclaje para distintas escalas.[28]

En lugar de predecir las coordenadas absolutas de los centros de los rectángulos delimitadores, YOLOv3 predice dos valores t_x, t_y, que son un desplazamiento relativo a las coordenadas de la celda de la cuadrícula, donde $(0, 0)$ es el vértice superior izquierdo de esa celda, y $(1, 1)$ es el vértice inferior derecho, por lo que aplica la función de activación sigmoide a las coordenadas del centro del rectángulo delimitador para que se mantengan en el rango de 0 a 1. De la comparación con las cajas de anclaje, YOLOv3 predice cuánto hay que reajustar las cajas de anclaje para que la diferencia en el alto y ancho con el rectángulo etiquetado (B_{GT}) sea mínima. Más precisamente, para cada celda y caja de

[27] Imagen adaptada de [79].

[28] Imagen adaptada de [93].

anclaje, la red predice el logaritmo de los factores de reajuste horizontal y vertical, que son t_w y t_h, respectivamente [76].

Si la celda está desplazada de la esquina superior izquierda de la imagen por (c_x, c_y) y la caja de anclaje tiene ancho p_w y alto p_h, entonces la red transforma las salidas t_x, t_y, t_w y t_h para dibujar los rectángulos predichos B_P de centro (b_x, b_y), ancho b_w y alto b_h, de acuerdo a [67]:

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e^{t_w} \\
b_h &= p_h e^{t_h},
\end{aligned}
$$

como se ilustra en la Figura 2.21.

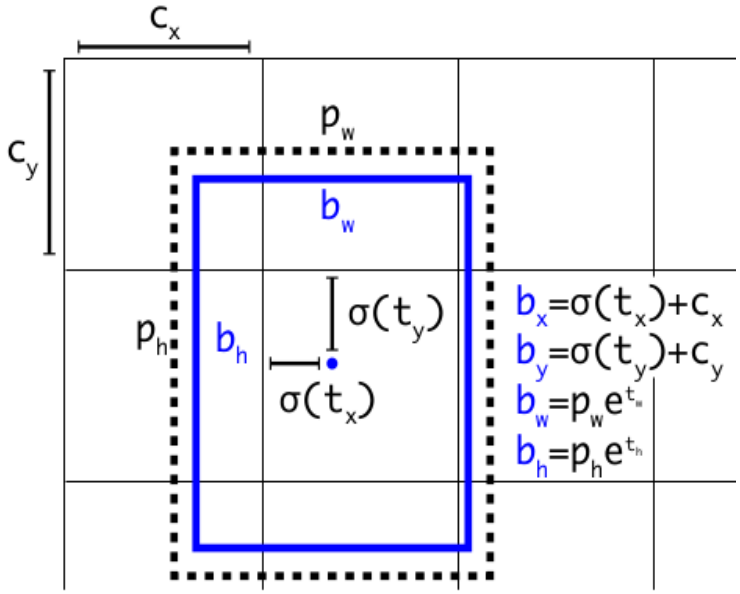


Figura 2.21. Rectángulo predicho (azul) y caja de anclaje (línea discontinua).[29]

Además, YOLOv3 usa una regresión logística para predecir un valor para cada rectángulo predicho, llamado *"objectness"*. Esta no es una palabra propia del diccionario de habla inglesa, por lo que no tiene una traducción directa, pero podemos pensarlo como "qué tan *objeto* es una detección dada". Este valor se interpreta como el nivel de confianza que algún objeto exista dentro de un rectángulo delimitador, el cual se espera sea 1 para la predicción que tiene mayor IoU con el rectángulo etiquetado [67]. Por otro lado, calcula una puntuación por clase, que es una probabilidad condicional (probabilidad que la detección sea de una clase determinada dado que existe un objeto en el rectángulo). Entonces, el nivel de confianza para cada clase es el producto entre el valor de *objectness* y la puntuación

[29]Imagen extraída de [67].

de la clase [41]. Si existen varias cajas delimitadoras que se superponen con el rectángulo etiquetado, entonces se hace uso de la supresión de no máximos, eligiendo la de mayor confianza e ignorando las predicciones cuya superposición sea mayor al valor umbral NMS determinado, el cual suele fijarse en 0,5 [89].

Cada celda predice $B = 3$ rectángulos, por lo que las salidas son tensores de tamaños $13 \times 13 \times (3 \cdot (5 + C))$, $26 \times 26 \times (3 \cdot (5 + C))$ y $52 \times 52 \times (3 \cdot (5 + C))$. En total YOLOv3 usa 9 cajas de anclaje (3 por escala).

Además de su estructura, y de la idea de "reciclar" las características extraídas de la escala grande para ayudarse a detectar objetos en las escalas mediana y pequeña, la red YOLOv3 se caracteriza por su función de costo a minimizar durante el entrenamiento. Hay quienes consideran que dicha función es la clave del éxito de esta red, pero paradójicamente no está formulada en [67]. Por ello, la explicación que sigue a continuación está basada en artículos de blogs y foros [79, 89, 58, 59, 77, 61, 71], siguiendo en parte el código [70], que mencionan modificaciones de la función de costo original de [41].

Función error

La función error consta de cuatro partes: error de las coordenadas del centro (2.6), error de las dimensiones alto y ancho (2.7), error de *objectness* (2.8) y error de clasificación (2.9):

$$\mathcal{E} = \lambda_{\text{coord}} \sum_{S\in\{13,26,52\}} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(t_x - \hat{t}_x)^2 + (t_y - \hat{t}_y)^2\right] \tag{2.6}$$

$$+ \lambda_{\text{coord}} \sum_{S\in\{13,26,52\}} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(t_w - \hat{t}_w)^2 + (t_h - \hat{t}_h)^2\right] \tag{2.7}$$

$$\begin{aligned} &+ \sum_{S\in\{13,26,52\}} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(-\mathscr{o} \ln(\hat{\mathscr{o}})\right) \\ &+ \lambda_{\text{noobj}} \sum_{S\in\{13,26,52\}} \sum_{i=1}^{S^2} \sum_{j=1}^{B} (1 - \mathbb{1}_{ij}^{\text{obj}})\, \xi \left(-(1 - \mathscr{o}) \ln(1 - \hat{\mathscr{o}})\right) \end{aligned} \tag{2.8}$$

$$+ \sum_{S\in\{13,26,52\}} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbb{1}_{ij}^{\text{obj}}\, \mathcal{E}_{BCE}(\mathcal{C}, \hat{\mathcal{C}}) \tag{2.9}$$

Para entender mejor esta función error, analizaremos cada uno de sus términos. Para ello, cabe recordar que cada celda predice B rectángulos distintos, por lo que se tiene en cuenta el error presente en cada celda. Además, la cantidad de celdas depende de la escala, siendo que hay tres escalas diferentes.

La primera parte (2.6) es la que corresponde al error de predicción del centro del rectángulo delimitador. Como se trata de un problema de regresión se usa la suma de los cuadrados del error, la cual calcula la diferencia entre los centros para cada celda, según

la escala. Sin embargo, no todas las celdas contienen un objeto, por lo que se agrega el factor $\mathbb{1}_{ij}^{\text{obj}}$ que es igual a 1 si en la celda hay un objeto real y 0 caso contrario. Además, se usa el factor λ_{coord} el cual tiene un valor de 5 y fue introducido en la primera versión de YOLO, con el objetivo de poner mayor énfasis en la localización que en la clasificación. El segundo término (2.7) corresponde al error de predicción del ancho y alto de los rectángulos delimitadores, y es calculado de forma similar al error de las coordenadas del centro [79].

La tercera parte (2.8) es el error de *objectness*. Este valor es una predicción de IoU: se interpreta como qué tan bien considera la red que el rectángulo B_P cubre el verdadero objeto B_{GT}. Este error enseña a la red a predecir un buen IoU, encontrando regiones de interés, mientras que (2.6) y (2.7) enseñan a predecir un mejor rectángulo (que eventualmente llevará al valor IoU a 1). Todas las cajas delimitadoras predichas contribuyen al error de *objectness*, pero sólo aquellos rectángulos que mejor se ajustan al objeto son los que contribuyen a los errores de centro, ancho y alto, y error de clasificación [89]. En YOLOv1 se usaba SSE para este error, pero esto fue cambiado en las siguientes versiones. Para el rectángulo etiquetado, el valor $\mathscr{o}$ de *objectness* es 1 para la celda que contiene un objeto y 0 caso contrario. Por otro lado, el segundo término de (2.8) penaliza las predicciones que no contienen un objeto, es decir las predicciones FP. El factor $\lambda_{\text{noobj}} = 0{,}5$ (introducido en la primera versión de YOLO) contribuye a no darle tanto énfasis a los FP, mientras que ξ es un factor que se usa para asegurar que sólo se penalice cuando la predicción no tenga mucho solapamiento con el rectángulo etiquetado [79]. Esto es, si una predicción no es la de mejor IoU con el rectángulo etiquetado, pero su IoU es mayor a un umbral (NMS = 0,5), entonces se ignora esa predicción (con el factor ξ) y por ende no se la tiene en cuenta en el cálculo del error [67]. Así, en estas situaciones se tiende a ser "más suave" porque en realidad la red está bastante cerca de la respuesta correcta. Caso contrario, es decir aquellas predicciones FP cuyo IoU sea menor al valor umbral, no contribuyen a los errores de coordenadas del centro, ancho, alto y clasificación, pero sí al error de *objectness* (en su segundo término).

Finalmente, (2.9) usa *binary cross-entropy* (Ecuación 2.5, Sección 2.2.2) para calcular el error propio de un problema de clasificación multi-etiqueta, donde $\mathcal{C}$ es el vector con todas las clases presentes en el conjunto de entrenamiento, y $\hat{\mathcal{C}}$ es el vector con tantos 1 como clases identificadas, y 0 en el resto de los lugares. Como antes, el factor $\mathbb{1}_{ij}^{\text{obj}}$ es para incluir el error de aquellas predicciones que se corresponden con una etiqueta real. Cabe destacar que en versiones anteriores no se calculaba de esta manera el error de clasificación, ya que el planteo original del problema era una clasificación multi-clase, que asume que las clases son mutuamente excluyentes. Pero como las bases de datos pueden tener más de una etiqueta de diferentes clases para un mismo objeto (por ejemplo, "persona" y "mujer"), se decidió cambiar el planteo por uno multi-etiqueta [79].

Presentaremos en el próximo Capítulo cómo fue la obtención de imágenes y las técnicas aplicadas a la base de datos para resolver el problema de detección de racimos de uva en fotografías de espalderos, usando la red YOLOv3.

Capítulo 3

Materiales y métodos

Como adelantamos en el Capítulo anterior, nos centraremos en resolver un problema de detección de objetos en imágenes, en este caso en particular se trata de racimos de uva. Para ello fue necesaria la búsqueda de una base de datos que tuviera imágenes de racimos. Al no dar con una, en primer lugar descargamos fotografías de internet, con las que hicimos las primeras pruebas. Sin embargo, como éstas no representan necesariamente la realidad mendocina, dado que no todas se corresponden con las variedades de la región, y como muchas de esas imágenes están protegidas por derechos de autor, procedimos a armar un conjunto de imágenes propio.

En este Capítulo presentamos entonces cómo fue la captura de imágenes con teléfonos celulares en distintas fincas, logrando un conjunto de más de 900 fotografías a color que etiquetamos manualmente usando una herramienta de `Python`, lo cual hemos detallado en la Sección 3.1. Entrenamos modelos de detección con conjuntos de entrenamiento de 100 y 500 fotografías, y luego aplicamos técnicas de aumento de datos a las imágenes del primer conjunto para observar si haciendo esto se obtienen resultados similares a entrenar con el segundo conjunto, para lo cual explicamos y ejemplificamos dichas técnicas en la Sección 3.2. Luego exponemos los valores de hiperparámetros (Sección 3.3), así como el hardware y software utilizados (Sección 3.4). Finalmente describimos en la Sección 3.5 los scripts de `Bash` y `Python` que utilizamos para el pre-procesamiento de las imágenes, el entrenamiento de la NN usando la biblioteca `ImageAI` y el post-análisis de los resultados obtenidos.

3.1. Imágenes

Un espaldero es una estructura armada a partir de postes y alambres sobre los que se dispone el cultivo y los toma como guía en su crecimiento. Se caracteriza por optimizar la intercepción de luz mejorando así el rendimiento y calidad de la uva, facilitar la mecanización para la poda y evitar enfermedades [19], por lo que es elegido por productores como sistema de conducción. Por ello, entre otras razones prácticas, decidimos armar una base de datos propia tomando fotografías de espalderos de diferentes fincas, incluyendo dos del Valle de Uco, y del Instituto Nacional de Tecnología Agropecuaria (INTA), en Luján de Cuyo. Las mismas fueron tomadas a finales de febrero y principios de marzo del 2020, en fechas próximas a Vendimia, cuando los racimos han alcanzado su máximo desarrollo. No fue necesario utilizar flash, ya que las capturamos durante la mañana, con iluminación natural. Utilizamos cámaras de teléfonos celulares de 13 y 25 megapixeles. Las variedades fotografiadas incluyen Malbec y Cabernet Sauvignon, ambas de color negro azulado (correspondiente a variedades tintas), usadas para elaborar vino tinto. Se hicieron primeros planos de los racimos y planos generales de los espalderos.

Luego procedimos a etiquetar cada imagen en el formato PASCAL VOC, de forma manual: cada imagen tiene asociado un archivo de igual nombre de extensión `.xml` que almacena las coordenadas de los vértices superior izquierdo e inferior derecho de cada etiqueta que se genera manualmente (considerando que el origen de coordenadas está en el extremo superior izquierdo de la fotografía), además de otros datos como el ancho y alto de la imagen. Para etiquetar usamos la herramienta de anotación de imágenes `LabelImg` [36], que está escrita en `Python`. Vale aclarar que las primeras pruebas que hicimos con imágenes de internet tenían por objetivo la detección de GRANOS de uva, pero luego en reuniones mantenidas entre el Director de este Seminario de Investigación con productores y técnicos del INTA y COVIAR, manifestaron que para ellos es más importante saber la cantidad de racimos que contar granos de uva. Esto provocó que cambiara el objetivo de la detección y pusiéramos el foco en analizar la capacidad de detección de RACIMOS de uva. En la Figura 3.1 se muestra un ejemplo del procedimiento de etiquetado usando esta herramienta, mientras que el archivo generado se muestra en el Código 3.1, donde podemos ver los vértices de los dos racimos etiquetados. Por ejemplo, el recuadro celeste en la Figura se corresponde con las coordenadas siguientes: vértice superior izquierdo $(x_{min}, y_{min}) = (1660, 1722)$, vértice inferior derecho $(x_{max}, y_{max}) = (2390, 2837)$.

```
<annotation>
  <folder>images</folder>
  <filename>bunch0823.jpg</filename>
  <path>/media/tatiana/Datos/Tati/tesis/propias/imagenes/bunch700/
    originales/validation/images/bunch0823.jpg</path>
  <source>
    <database>Unknown</database>
  </source>
```

```
  <size>
    <width>4160</width>
    <height>3120</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>bunch</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>2100</xmin>
      <ymin>1067</ymin>
      <xmax>2735</xmax>
      <ymax>2177</ymax>
    </bndbox>
  </object>
  <object>
    <name>bunch</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>1660</xmin>
      <ymin>1722</ymin>
      <xmax>2390</xmax>
      <ymax>2837</ymax>
    </bndbox>
  </object>
</annotation>
```

Código 3.1. Archivo .xml generado al etiquetar usando LabelImg.

Etiquetamos aproximadamente 900 imágenes, que se corresponden con más de 3000 racimos. Luego separamos estas imágenes y sus respectivas etiquetas en cuatro conjuntos, que llamamos *bunch*600.1, *bunch*600.2, *bunch*120 y *detect*, los cuales se detallan a continuación:

bunch600 consiste en 500 imágenes usadas como conjunto de entrenamiento $\mathcal{D}_{train}$, y 100 imágenes usadas para la etapa de validación $\mathcal{D}_{val}$. En estos conjuntos hay fotografías tomadas en primer plano y en plano general. La diferencia entre *bunch*600.1 y *bunch*600.2 radica en que se cambiaron las imágenes de *bunch*600.1 tomadas a mayor distancia de los espalderos por otras capturadas a menor distancia, a fin de que los racimos se vieran en mayor tamaño. Así, el primero de estos cuenta con 2253 racimos etiquetados en $\mathcal{D}_{train}$ y 472 en $\mathcal{D}_{val}$, mientras que el segundo tiene 2393 racimos etiquetados en $\mathcal{D}_{train}$ y 349 en $\mathcal{D}_{val}$. Los resultados obtenidos a partir del entrenamiento con *bunch*600 se tomaron como punto de comparación con el resto de

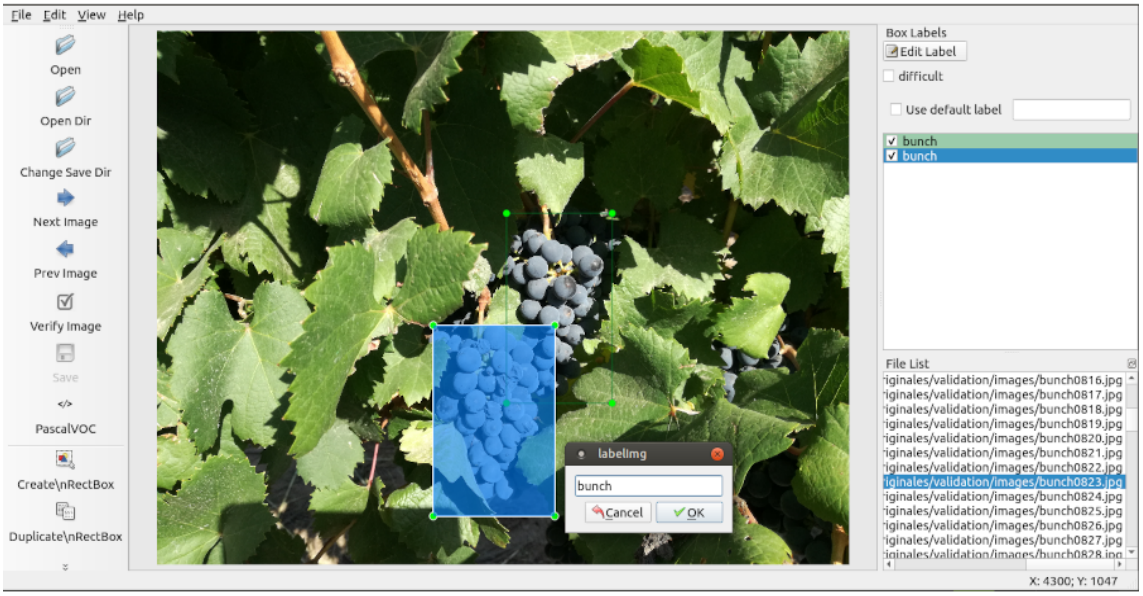


Figura 3.1. Interfaz gráfica de LabelImg, ejemplo de imagen etiquetada.

los análisis, ya que al ser un dataset más numeroso se espera que logre mayores y mejores detecciones;

bunch120 consiste de 100 imágenes usadas como conjunto de entrenamiento $\mathcal{D}_{train}$ (593 racimos etiquetados), y 20 imágenes usadas para la etapa de validación $\mathcal{D}_{val}$ (102 racimos etiquetados). En este caso todas las imágenes usadas fueron tomadas en primer plano;

detect consiste en 100 imágenes (641 racimos etiquetados) en $\mathcal{D}_{test}$, que fueron utilizadas posteriormente al entrenamiento, para evaluar qué tan bien son detectados racimos que son completamente nuevos para la red.

Hicimos pruebas tanto con *bunch*120 como con *bunch*600. Luego, conscientes de que el primero de estos cuenta con muy pocas fotografías, y como para resolver problemas de detección de objetos se requieren en general de cientos a miles de imágenes, hicimos uso de técnicas de aumento de datos que aplicamos a las fotografías de *bunch*120 para obtener conjuntos de entrenamiento más numerosos, y ver si haciendo esto se pueden obtener resultados similares a los logrados con *bunch*600.

3.2. Técnicas de aumento de datos

El AUMENTO DE DATOS presentes en el dataset (para el caso de detección de objetos en imágenes se conoce como *image augmentation*) es una técnica usual que busca incrementar la cantidad de imágenes cuando se tiene un conjunto de entrenamiento escaso [63, 84]. Este método se fundamenta en cómo las NN interpretan las características de la imagen: pequeñas modificaciones en sus píxeles resultan en imágenes diferentes para la NN. Por ejemplo,

en [56] se usa la ecualización adaptativa de histograma limitado por contraste (CLAHE, por sus siglas en inglés *Contrast Limited Adaptive Histogram Equalization*), mientras que en [86] se espejan las imágenes y se aplica un efecto de empañado (*blurring*) en problemas de estimación de cosecha de uva. Por otro lado, como se explica en [84] otras transformaciones utilizadas son el recortado, rotación, traslación, agregado de ruidos (consiste en sumar una matriz de valores aleatorios extraídos usualmente de una distribución normal), así como transformaciones en uno o más de los canales de color de la imagen.

Motivados por los ejemplos anteriores, aplicamos una serie de filtros y transformaciones a las imágenes de *bunch*120, que es el dataset con menor número de racimos, usando la herramienta `ImageMagick` [99]. Vale aclarar que quisimos aplicar las mismas transformaciones al conjunto *bunch*600, sin embargo por cuestiones de tiempo esto no fue posible, como veremos en el Capítulo 4. Por un lado, llevamos a cabo pruebas en las que se DUPLICÓ el conjunto de entrenamiento y validación, usando la transformación de reflexión, el filtro de solarización y CLAHE:

reflexión: espejado horizontal respecto a la recta $x = C_x$, donde (C_x, C_y) es el centro de la imagen [105]. En la Figura 3.2 se muestra un ejemplo de una imagen original, es decir sin la aplicación de ningún tipo de filtro, junto a su reflexión;

(a) Original (b) Reflexión

Figura 3.2. Ejemplo de una imagen original y de su reflexión.

solarización: al solarizar se invierte el color de los píxeles cuya luminosidad sobrepase un cierto límite [106, 112], en este caso se fijó el límite en 60 %. Este filtro produce el efecto que se observa al exponer una película fotográfica a la luz durante el proceso de revelado;

CLAHE: la ecualización adaptativa de histogramas mejora el contraste local y potencia las definiciones de los bordes en cada región de una imagen. Para aplicar CLAHE se usa una cuadrícula de celdas de ancho y alto de píxel definidos, donde cada recuadro contiene un número fijo de clases o bins, y se establece un límite para los cambios localizados de contraste [104]. En este caso los recuadros fueron del 5 % del ancho y

alto de la imagen, y contenían 128 bins cada uno. Además se fijó el límite de contraste en 4.

En la Figura 3.3 se muestra el efecto de aplicar solarización (b) y CLAHE (c) a la imagen de la Figura 3.2(a) (vista del racimo izquierdo).

(a) Original (racimo izquierdo Figura 3.2(a)) (b) Solarización (c) CLAHE

Figura 3.3. Efecto de los filtros de solarización y CLAHE aplicados a la imagen de la Figura 3.2(a).

Por otro lado, realizamos pruebas en las que se QUINTUPLICÓ el conjunto de entrenamiento y validación al aplicar cuatro valores distintos del mismo tipo de filtro:

ruidos: además del agregado de ruido Gaussiano, hicimos pruebas donde se sumaron valores extraídos de una distribución de Poisson, y de Laplace [115]. La cantidad de ruido a añadir puede ser controlada por el valor `-attenuate`. Usamos los valores de atenuación 1,5; 3; 4,5 y 6 para cada tipo de ruido;

paint: esta técnica consiste en aplicar un difuminado a partir de "gotas" de pintura que fusiona los colores presentes en un entorno y los convierte en un área de un solo color [113]. Usamos los valores de radio 3, 5, 7 y 9;

empañado: para aplicar este filtro los parámetros a tener en cuenta son `{radius}x{sigma}` [111]. El primer valor (radio) controla el tamaño del área donde se van a difuminar los píxeles. Debe ser como mínimo el doble de sigma, o puede utilizarse el valor 0 para el cual el algoritmo determina cuál es el radio adecuado para el valor de sigma dado. A su vez, sigma puede considerarse como una aproximación de cuánto se espera que se "extienda" o difumine la imagen, en píxeles. Usamos los valores $0 \times 2, 0 \times 5, 0 \times 8$ y 0×11.

En la Figura 3.4 se observa un difuminado por "gotas" de pintura y el efecto de empañar la imagen de la Figura 3.2(a). A su vez, en la Figura 3.5 se muestra el efecto de aplicar ruido Gaussiano, de Poisson y Laplaciano a la misma imagen.

(a) Original (racimo izquierdo Figura 3.2(a)) (b) Paint, radio 9 (c) Empañado, sigma 11

Figura 3.4. Efecto de los filtros pintura y empañado aplicados a la imagen de la Figura 3.2(a).

(a) Original (racimo izquierdo Figura 3.2(a)) (b) Ruido Gaussiano, atenuación de 3 (c) Ruido de Poisson, atenuación de 1,5 (d) Ruido Laplaciano, atenuación de 6

Figura 3.5. Efecto del agregado de distintos ruidos a la imagen de la Figura 3.2(a).

Todos los filtros utilizados modifican las imágenes, pero no las etiquetas de los racimos, por lo que su aplicación en la mayoría de los casos permite usar las anotaciones existentes y solo se requiere cambiar el nombre de la imagen en la línea `<filename>` del archivo `.xml` (línea 3 del ejemplo mostrado en el Código 3.1). En cambio, la reflexión requiere que se aplique también una transformación adecuada en los archivos `.xml` para que las etiquetas tengan los vértices espejados. En particular, la posición vertical de los vértices superior izquierdo e inferior derecho dadas por la segunda componente y_{min} e y_{max}, respectivamente (líneas 21, 23, 33, 35 del ejemplo presente en Código 3.1), permanece sin modificaciones, mientras que la primera componente se ve alterada: si llamamos x a la posición horizontal del vértice etiquetado (líneas 20, 22, 32, 34 del ejemplo presente en Código 3.1), y x^E a la posición del vértice espejado, entonces usando el dato del ancho a de la imagen (línea 9 del

ejemplo presente en Código 3.1), se obtiene que

$$(x^E_{min},\, y^E_{min}) = (a - x_{max},\, y_{min})$$
$$(x^E_{max},\, y^E_{max}) = (a - x_{min},\, y_{max})$$

Así a partir de las 100 imágenes de *bunch*120 obtuvimos nuevos conjuntos de entrenamiento, los primeros de 200 imágenes cada uno, los segundos de 500 imágenes cada uno, igualando éstos en número de imágenes a $\mathcal{D}_{train}$ de *bunch*600. A su vez, aplicamos las mismas transformaciones a las imágenes de validación, para mantener las proporciones entre ambos conjuntos. Con estos datos entrenamos redes neuronales que usan la arquitectura de YOLOv3, combinando distintos valores de hiperparámetros, los cuales detallamos a continuación.

3.3. Especificaciones técnicas del software utilizado

Para llevar a acabo los diferentes análisis usamos la biblioteca `ImageAI` [64] de `Python`, ya que permite entrenar modelos personalizados de detección de objetos, es decir con una o más clases de objetos propias (en este caso, racimos de uva), sobre conjuntos de imágenes cuyo formato de etiquetas sea el de PASCAL VOC. `ImageAI` usa la arquitectura YOLOv3 [41, 67], y solo requiere unas pocas líneas de código. Con esta biblioteca llevamos a cabo tres grandes etapas: entrenamiento y validación, evaluación de los modelos generados durante el entrenamiento, y puesta a prueba del mejor modelo con el conjunto de datos *detect*.

Con `ImageAI` entrenamos una red neuronal distinta para cada conjunto de entrenamiento $\mathcal{D}_{train}$ (ya sea alguno de *bunch*600, el de *bunch*120 o el formado a partir de las imágenes de *bunch*120 junto con alguna de las distorsiones explicadas en la Sección 3.2). Durante el entrenamiento se van ajustando los parámetros en cada época, generando distintos modelos de entrenamiento. Sin embargo no todos ellos son almacenados y considerados para la etapa de evaluación, sino solamente aquellos que logren disminuir el error de entrenamiento de una época a la otra, por lo que generalmente no se obtienen tantos modelos como número de experimentos, sino menos. Entonces, al finalizar el entrenamiento, evaluamos con `ImageAI` los distintos modelos almacenados, calculando el valor de mAP de cada uno. Finalmente seleccionamos el modelo de mayor mAP y lo ponemos a prueba con $\mathcal{D}_{test}$ (es decir, el conjunto *detect*).

Durante las primeras pruebas de detección de racimos con imágenes descargadas de internet, llevamos a cabo el entrenamiento desde cero. Sin embargo advertimos que sin importar la elección de hiperparámetros, en la etapa de evaluación todos los modelos tenían un mAP de 0, probablemente debido a que no contamos con un gran número de datos en nuestro dataset. Es por ello que para la etapa de entrenamiento decidimos utilizar un modelo pre-entrenado facilitado por la misma biblioteca `ImageAI`, es decir, nos beneficiamos así de la transferencia de aprendizaje.

Respecto a los hiperparámetros utilizados, hicimos pruebas para entrenar con 50 y con 100 épocas (*epoch*), con tamaños de lote (*batch size*) de $2, 4, 8$ y 12, ya que la memoria de las GPUs con las que contamos (descriptas en la siguiente Sección) no era suficiente para entrenar con lotes de mayor tamaño. También hicimos pruebas con tasa de aprendizaje (*learning rate*) de 1×10^{-4}, 5×10^{-5} y 1.5×10^{-4}. Para la etapa de evaluación usamos IoU = 0,5, OBJ = 0,3 y NMS = 0,5 en todas las pruebas. Finalmente, para la puesta a prueba de los modelos, reportamos aquellas detecciones cuyo nivel de certeza fuera mayor o igual a 0,3.

3.4. Descripción de Hardware y Software utilizado

Todo el proceso de análisis de las imágenes (entrenamiento, validación, evaluación y detección) fue realizado principalmente utilizando GPUs (*Graphics Processing Units*). Utilizamos el siguiente hardware perteneciente al cluster Toko ubicado en FCEN-UNCUYO:

- Nodo de cálculo de Toko con cuatro AMD **Opteron** 6376 CPU, con 16 núcleos CPU de 2.3GHz cada uno (64 núcleos en total) y 128 GB de RAM.
- Nodo de cálculo de Toko con dos AMD **EPYC** 7281 CPU, con 16 núcleos CPU de 2.1GHz cada uno (32 núcleos en total), 128 GB de RAM.
- Nodo de cálculo de Toko con un AMD **Ryzen** 7 2700 con 8 núcleos y 16 hilos de 3.2 GHz con 64 GB de memoria RAM DDR4.
- Dos nodos AMD FX-8350 con 8 núcleos de 4 GHz y 32 GB de RAM DDR3. Un nodo con una GPU NVIDIA GeForce GTX **Titan X** (arquitectura Maxwell GM200) con 12 GB de memoria, y el otro nodo con una GPU NVIDIA GeForce GTX **Titan Xp** (arquitectura Pascal GP102) con 12 GB de memoria.

En cuanto al software que el cluster Toko utiliza, tiene instalado Slackware Linux Current 2020 de 64 bits con kernel 5.4.28, gcc 9.2, NVIDIA driver 410.73 y CUDA 10. El software utilizado para realizar el análisis de las imágenes se desarrolló con la biblioteca `ImageAI` 2.1.5 con `tensorflow-gpu` 1.13.1, `keras` 2.2.4, y `Python` 3.7.2. En la Sección 4.6 se encuentran los resultados del análisis de rendimiento de la NN sobre este hardware.

3.5. Descripción de los scripts utilizados

`ImageAI` requiere un orden especial de los archivos para acceder a los mismos durante las distintas etapas: cada entrenamiento requiere el acceso a un directorio con carpetas nombradas *train* y *validation*, que almacenan las imágenes y anotaciones usadas para el entrenamiento y validación, respectivamente, en carpetas llamadas *images* y *annotations*. Como cada vez que se comienza la etapa de entrenamiento `ImageAI` crea automáticamente

carpetas específicas para almacenar datos de la red, en particular los modelos generados, es necesaria la previa creación de un directorio diferente para cada prueba a ejecutar, ya que si no se hace esto las carpetas que usa `ImageAI` se sobrescribirían y se perdería la información antes almacenada. Es por ello que aprendí comandos básicos de `Bash` que me permitieron manipular directorios y grandes cantidades de archivos. Para el pre-procesamiento de las imágenes, aprendí a escribir scripts en este lenguaje que crearon carpetas para cada filtro, y mediante un bucle `for` aplicaron el filtro o transformación correspondiente a cada imagen así como la edición de los archivos `.xml` para modificar las líneas necesarias. Ejemplos de estos scripts se pueden consultar en el Apéndice A.

Luego, escribí un script de `Bash` donde definí los hiperparámetros a utilizar (explicados en la Sección 3.3), la creación de directorios correspondientes, y la ejecución de un script de `Python` pasándole todos los valores de los hiperparámetros definidos. En dicho script de `Python` importé las bibliotecas `os`, `numpy` [17], `time` y `sys`, así como las funciones `DetectionModelTrainer` y `CustomObjectDetection` de `imageai.Detection.Custom`. Además definí nuevas funciones que llevan a cabo el entrenamiento, la evaluación de los modelos cuando termina con el entrenamiento y, una vez seleccionado el modelo, lo pone a prueba con el conjunto de datos *detect*. Un ejemplo de estos scripts se encuentra en el Apéndice B. En caso de entrenar con muchas imágenes, a menudo separaba estas etapas ejecutando tres scripts por separado, para no ocupar tanta memoria de las GPUs.

Finalmente, armé un script para contabilizar la cantidad de detecciones, hacer gráficas y dibujar los rectángulos etiquetados y los detectados en cada imagen, a fin de tener una idea visual de la calidad de las detecciones, el cual se puede ver en el Apéndice C.

En el siguiente Capítulo exponemos los principales resultados y discusiones obtenidos del entrenamiento con los diferentes conjuntos, así como con los distintos valores de hiperparámetros utilizados.

Capítulo 4

Resultados y Discusión

El objetivo de este capítulo es mostrar los principales resultados obtenidos, en el orden en que fuimos desarrollando las distintas pruebas según los interrogantes que surgieron. Los problemas de detección de objetos en imágenes requieren miles a cientos de miles de imágenes para la etapa de entrenamiento. Sin embargo nosotros no contamos con tal cantidad de fotografías, ya que tuvimos acceso limitado en el tiempo a las fincas. Es por esto que tomamos como dataset de referencia a *bunch*600, ya que es el conjunto de datos más numeroso que tenemos. Además, el trabajar con una mayor cantidad de imágenes requeriría mucho tiempo de cómputo, lo cual se dificulta especialmente al no disponer de suficiente hardware para llevar a cabo el entrenamiento de la red, ya que éste es compartido con otros usuarios.

Así, tomamos como rendimiento de referencia de la NN el alcanzado al entrenar con *bunch*600 para comparar con el resto de las pruebas. Para ello graficamos el error de entrenamiento y validación alcanzado luego de cada época y expusimos algunas medidas de rendimiento como exhaustividad, precisión o mAP. Además hicimos pruebas con el dataset *bunch*120 que tiene una menor cantidad de fotografías, al cual le aplicamos diferentes técnicas de aumento de datos, y también lo usamos para comparar el comportamiento de la NN usando diferentes hiperparámetros e infraestructura de hardware.

Comenzamos describiendo las salidas obtenidas del uso de `ImageAI` en la Sección 4.1 y explicando el criterio utilizado para el cálculo de exhaustividad y precisión en la Sección 4.2. Luego presentamos los resultados y algunas conclusiones principales de la comparación del entrenamiento con *bunch*600 y con *bunch*120, así como con el uso de técnicas de aumento de datos en la Sección 4.3. Además, en la Sección 4.4 presentamos los resultados de utilizar diferentes hiperparámetros, y en la Sección 4.5 analizamos algunos interrogantes relacionados a la aleatoriedad en la elección de las imágenes. Finalmente, como interés secundario presentamos el rendimiento computacional de la NN en diferentes hardware en la Sección 4.6.

4.1. Descripción de las salidas

Para ejemplificar lo obtenido luego de correr los scripts descriptos en la Sección 3.5, el Código 4.1 muestra las primeras líneas que se imprimen al comenzar la etapa de entrenamiento. Vemos que entre las líneas 28 y 30 genera las cajas de anclaje descriptas en la Sección 2.5.3. En la línea 32 nombra en qué clases será entrenada la red, en este caso es sólo una: racimos de uva (etiquetados como `'bunch'`). De las líneas 33 a la 35 fija los hiperparámetros tamaño de lote y número de experimentos o épocas, y aplica transferencia de aprendizaje, usando un modelo pre-entrenado facilitado por la biblioteca `ImageAI`. Luego, en la línea 36 comienza el entrenamiento con la primera época (*epoch*), para lo cual, en este ejemplo, se iterará 400 veces hasta completar el conjunto de entrenamiento. La cantidad de iteraciones la determina automáticamente `ImageAI`, y para cada iteración imprime el error (`loss`) de entrenamiento total, así como las contribuciones al error correspondientes a cada una de las capas YOLO encargadas de la detección en las distintas escalas (capas 83, 95 y 107 del Cuadro 2.3). Finalmente, en el Código 4.2 se muestran las últimas líneas que se imprimen al finalizar una época. En particular observamos que antes de comenzar el siguiente experimento, además del error de entrenamiento, imprime el error correspondiente a la etapa de validación (`val_loss` en la línea 437).

```
27 Using TensorFlow backend.
28 Generating anchor boxes for training images and annotation...
29 Average IOU for 9 anchors: 0.80
30 Anchor Boxes generated.
31 Detection configuration saved in  /home/tparlanti/JAIIO/answers/bunch120/
       originales/originales-100_experimentos/json/detection_config.json
32 Training on:  ['bunch']
33 Training with Batch Size:  2
34 Number of Experiments:  100
35 Training with transfer learning from pretrained Model
36 Epoch 1/100
37
38   1/400 [..............................] - ETA: 4:16:38 - loss: 109.8232 -
       yolo_layer_1_loss: 20.1749 - yolo_layer_2_loss: 26.4171 -
       yolo_layer_3_loss: 63.2311
39   2/400 [..............................] - ETA: 2:11:24 - loss: 109.8003 -
       yolo_layer_1_loss: 20.5621 - yolo_layer_2_loss: 26.3157 -
       yolo_layer_3_loss: 62.9225
40   3/400 [..............................] - ETA: 1:29:39 - loss: 108.6983 -
       yolo_layer_1_loss: 20.2209 - yolo_layer_2_loss: 25.5189 -
       yolo_layer_3_loss: 62.9586
41   4/400 [..............................] - ETA: 1:08:47 - loss: 107.1618 -
       yolo_layer_1_loss: 19.6580 - yolo_layer_2_loss: 24.8615 -
       yolo_layer_3_loss: 62.6423
```

Código 4.1. Primeras líneas impresas al comenzar el entrenamiento de la red

```
398/400 [============================>.] - ETA: 2s - loss: 33.0548 -
    yolo_layer_1_loss: 5.9282 - yolo_layer_2_loss: 7.6729 -
    yolo_layer_3_loss: 19.4537
399/400 [============================>.] - ETA: 1s - loss: 33.0220 -
    yolo_layer_1_loss: 5.9240 - yolo_layer_2_loss: 7.6653 -
    yolo_layer_3_loss: 19.4328
400/400 [==============================] - 489s 1s/step - loss: 32.9948 -
    yolo_layer_1_loss: 5.9164 - yolo_layer_2_loss: 7.6550 -
    yolo_layer_3_loss: 19.4234 - val_loss: 20.2140 - val_yolo_layer_1_loss:
     4.1390 - val_yolo_layer_2_loss: 5.3155 - val_yolo_layer_3_loss:
    10.7596
Epoch 2/100

  1/400 [..............................] - ETA: 6:58 - loss: 18.0877 -
    yolo_layer_1_loss: 2.5299 - yolo_layer_2_loss: 2.6744 -
    yolo_layer_3_loss: 12.8834
```

Código 4.2. Últimas líneas impresas al finalizar una época

Luego, en la etapa de evaluación de los modelos se imprimen la ruta al modelo, así como los hiperparámetros elegidos para esta etapa y el valor de mAP. Por ejemplo, en el Código 4.3 vemos la evaluación de los modelos generados al finalizar las épocas 19 (línea 40383), 20 (línea 40391), 23 (línea 40399) y 24 (línea 40407). El nombre de cada modelo incluye el número de experimento, así como el error de entrenamiento para esa época, por lo que podemos ver que para el experimento 20 el error fue de 8,223, mientras que para la época 23 el error fue de 7,475. De hecho, si consultamos el error de entrenamiento al finalizar los experimentos 21 y 22 vemos que fueron de 8,371 y 8,227, ambos por encima del error de la época 20. Es por esto que durante el entrenamiento no se generaron modelos para las épocas 21 y 22, ya que el objetivo es tener el mejor modelo que contribuya a disminuir el error, pero que tenga un buen valor de mAP.

```
Model File:  /home/tparlanti/JAIIO/answers/bunch120/originales/originales
    -100_experimentos/models/detection_model-ex-019--loss-0008.438.h5

Using IoU :  0.5
Using Object Threshold :  0.3
Using Non-Maximum Suppression :  0.5
bunch: 0.5729
mAP: 0.5729
================================
Model File:  /home/tparlanti/JAIIO/answers/bunch120/originales/originales
    -100_experimentos/models/detection_model-ex-020--loss-0008.223.h5

Using IoU :  0.5
Using Object Threshold :  0.3
Using Non-Maximum Suppression :  0.5
bunch: 0.5891
mAP: 0.5891
```

```
40398 ================================
40399 Model File:  /home/tparlanti/JAIIO/answers/bunch120/originales/originales
      -100_experimentos/models/detection_model-ex-023--loss-0007.475.h5
40400
40401 Using IoU :  0.5
40402 Using Object Threshold :  0.3
40403 Using Non-Maximum Suppression :  0.5
40404 bunch: 0.5675
40405 mAP: 0.5675
40406 ================================
40407 Model File:  /home/tparlanti/JAIIO/answers/bunch120/originales/originales
      -100_experimentos/models/detection_model-ex-024--loss-0006.743.h5
40408
40409 Using IoU :  0.5
40410 Using Object Threshold :  0.3
40411 Using Non-Maximum Suppression :  0.5
40412 bunch: 0.5517
40413 mAP: 0.5517
```

Código 4.3. Evaluación de modelos

Finalmente, una vez seleccionado el modelo de mayor mAP, lo ponemos a prueba con las imágenes del conjunto *detect* y se obtiene una salida como la que se muestra en el Código 4.4. En este ejemplo se eligió el modelo generado en la época 20, y se reportan, por imagen, los niveles de certeza de las detecciones (mayores a 30 %), así como las coordenadas de los vértices superior izquierdo (x_{min}, y_{min}) e inferior derecho (x_{max}, y_{max}) de los rectángulos delimitadores, en una lista de `Python` (en este caso, un vector) de componentes $[x_{min}, y_{min}, x_{max}, y_{max}]$. Si bien YOLOv3 reporta las coordenadas del centro, ancho y alto de cada detección, `ImageAI` realiza una transformación interna para reportar las coordenadas de los vértices como mencionamos antes.

```
40467  usando  /home/tparlanti/JAIIO/answers/bunch120/originales/originales-100
       _experimentos/models/detection_model-ex-020--loss-0008.223.h5
40468 bunch0717.jpg  :  bunch  :  47.45146334171295  :  [489, 1343, 1205, 2210]
40469 bunch0717.jpg  :  bunch  :  58.26789736747742  :  [267, 3154, 1243, 4078]
40470 bunch0717.jpg  :  bunch  :  33.16583335399628  :  [419, 2438, 860, 3095]
40471  usando  /home/tparlanti/JAIIO/answers/bunch120/originales/originales-100
       _experimentos/models/detection_model-ex-020--loss-0008.223.h5
40472 bunch0706.jpg  :  bunch  :  51.8044650554657  :  [1866, 1771, 2437, 2872]
40473  usando  /home/tparlanti/JAIIO/answers/bunch120/originales/originales-100
       _experimentos/models/detection_model-ex-020--loss-0008.223.h5
40474 bunch0780.jpg  :  bunch  :  58.34740996360779  :  [2927, 435, 3388, 1364]
40475 bunch0780.jpg  :  bunch  :  40.45214056968689  :  [1242, 1062, 1758, 1894]
40476 bunch0780.jpg  :  bunch  :  37.403157353401184  :  [1803, 1314, 2265, 2208]
40477 bunch0780.jpg  :  bunch  :  32.56012797355652  :  [3965, 464, 4127, 796]
40478 bunch0780.jpg  :  bunch  :  35.502687096595764  :  [2132, 413, 2426, 1072]
40479 bunch0780.jpg  :  bunch  :  54.93343472480774  :  [3685, 1453, 4067, 1807]
```

Código 4.4. Detecciones del modelo

Como `ImageAI` no reporta la matriz de confusión ni otras medidas de rendimiento en la etapa de detección, nos vimos en la necesidad de estimar alguna de ellas para poder comparar el desempeño de los modelos seleccionados. A continuación presentamos las medidas de rendimiento elegidas y el criterio empleado para decidir si un racimo etiquetado fue o no detectado correctamente.

4.2. Estimación de exhaustividad y precisión

Accedimos a los archivos `.xml` con las etiquetas de cada fotografía de *detect*, así como al archivo `.txt` con las salidas obtenidas, para tomar la información de los vértices y el porcentaje de certeza y hacer las comparaciones correspondientes. Como sólo tenemos una clase de objetos, todas las detecciones encontradas pertenecen a ésta, es decir, son consideradas "racimos", por lo que descartamos la posibilidad que un racimo sea detectado como otra clase. Sin embargo, existen racimos que no son detectados, así como racimos que son detectados más de una vez a pesar de la supresión de no máximos, debido a que a menudo un racimo grande es fraccionado en partes, especialmente cuando existen ramas u hojas que lo obstruyen parcialmente. Pudimos determinar visualmente que YOLO detectaba correctamente muchos racimos con un nivel de certeza bajo, esto se debe a que entrenamos con pocas imágenes, por lo que decidimos reportar todas las detecciones que tuvieran un nivel de certeza mayor o igual a 30 %, siendo que fijamos ese valor con un hiperparámetro al comenzar la etapa de detección.

Las medidas de rendimiento que elegimos fueron la exhaustividad (*recall*) para cuantificar cuántos racimos etiquetados fueron detectados por los modelos y determinar entonces los porcentaje de racimos detectados correctamente, así como la precisión (*precision*) para medir la exactitud de los modelos para detectar racimos. Como explicamos en la Sección 3.1, *detect* es el conjunto de imágenes que usamos para poner a prueba los modelos. Consta de 100 imágenes, que contienen 641 racimos etiquetados. Estas imágenes fueron etiquetadas con el fin de compararlas contra las detecciones de la NN, pero en ningún momento del entrenamiento la misma tuvo acceso a las imágenes ni a sus etiquetas. De cada rectángulo, ya sea etiquetado como detectado, sólo contamos con la información de los vértices superior izquierdo e inferior derecho, pero no podemos determinar si una detección se corresponde con una etiqueta por una mera comparación de dichos vértices, ya que es improbable que éstos coincidan. Como la mayoría de las fotografías tiene más de un racimo, para poder determinar qué detección o detecciones se correspondían con un rectángulo etiquetado, escribí un script en `Python` para comparar cada detección encontrada en una fotografía contra cada racimo etiquetado en la misma. Visualmente, dadas las características de las detecciones, establecí el criterio que se explica a continuación. En primer lugar:

1. Dado un rectángulo etiquetado, si la intersección entre éste y un rectángulo detectado representa más del 25 % de la superficie del rectángulo etiquetado, entonces los datos

de dicho rectángulo detectado se almacenan como POSIBLE DETECCIÓN del racimo etiquetado en cuestión. Esto lo hice para evitar comparaciones innecesarias entre esa detección con otros racimos etiquetados alejados dentro de la misma imagen. Por ejemplo, en la Figura 4.1 vemos en color rojo los racimos etiquetados para esa fotografía y en azul los racimos detectados por la red.

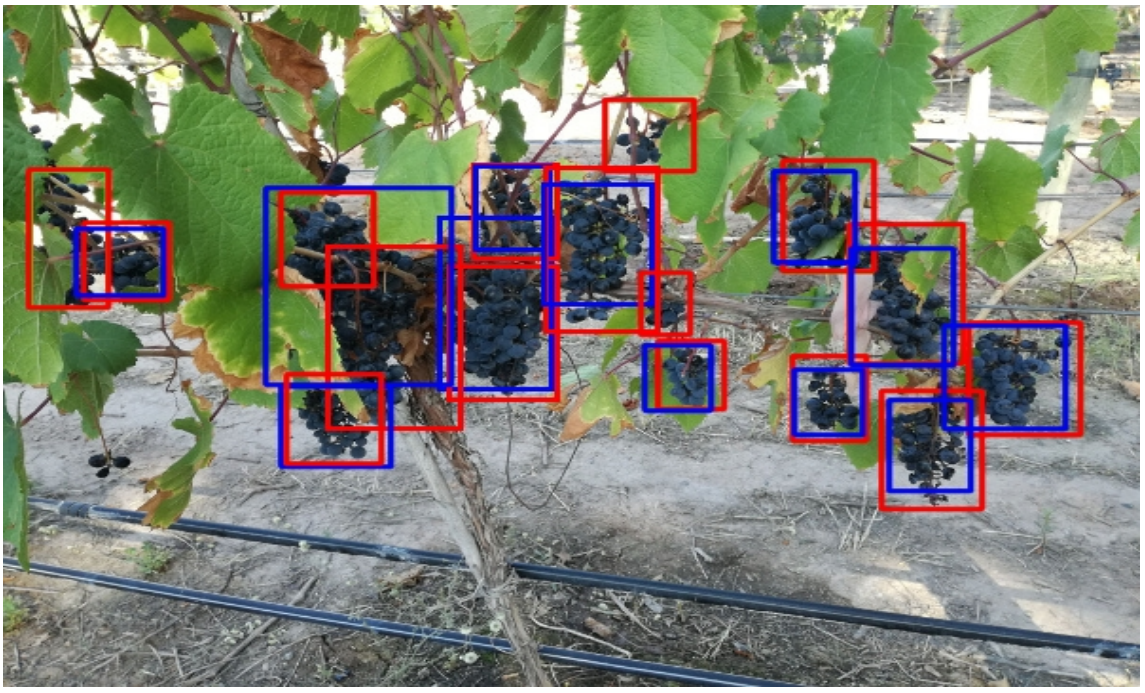


Figura 4.1. Ejemplo de racimos etiquetados (rojo) y detectados (azul). Para comparar ambos tipos de rectángulos sólo miramos aquellos que se intersectan en al menos un 25 %.

Para determinar si la detección de un racimo etiquetado fue exitosa, me concentré únicamente en las detecciones almacenadas para ese rectángulo etiquetado:

2. Si para un racimo etiquetado se ha encontrado sólo una posible detección, y si la intersección entre estos rectángulos representa al menos el 60 % de la superficie del rectángulo etiquetado, entonces consideré que ese racimo etiquetado fue DETECTADO CORRECTAMENTE. Por ejemplo, en la Figura 4.2 vemos que todos los racimos etiquetados fueron detectados correctamente.

3. En el caso que para un racimo etiquetado exista más de una detección posible, consideré que fue DETECTADO CORRECTAMENTE si la suma de las intersecciones entre cada detección con el rectángulo etiquetado fuese de al menos el 60 % de la superficie de este último. Por ejemplo, en la Figura 4.3 vemos que el racimo central etiquetado fue fraccionado en dos detecciones debido a que una hoja lo obstruye parcialmente.

4. Si para un racimo etiquetado no existen posibles detecciones, o si éstas no satisfacen 2 ó 3, entonces consideré que ese racimo NO FUE DETECTADO por el modelo. Por

ejemplo, en la Figura 4.4 vemos que los tres racimos superiores etiquetados no fueron detectados. Además, vemos que la detección de la izquierda tiene intersección superior al 25 % con dos rectángulos etiquetados. Sin embargo, como la intersección con el racimo etiquetado de arriba no supera el 60 % se considera no detectado. Así, en esta imagen de los seis racimos etiquetados solo dos fueron detectados correctamente.

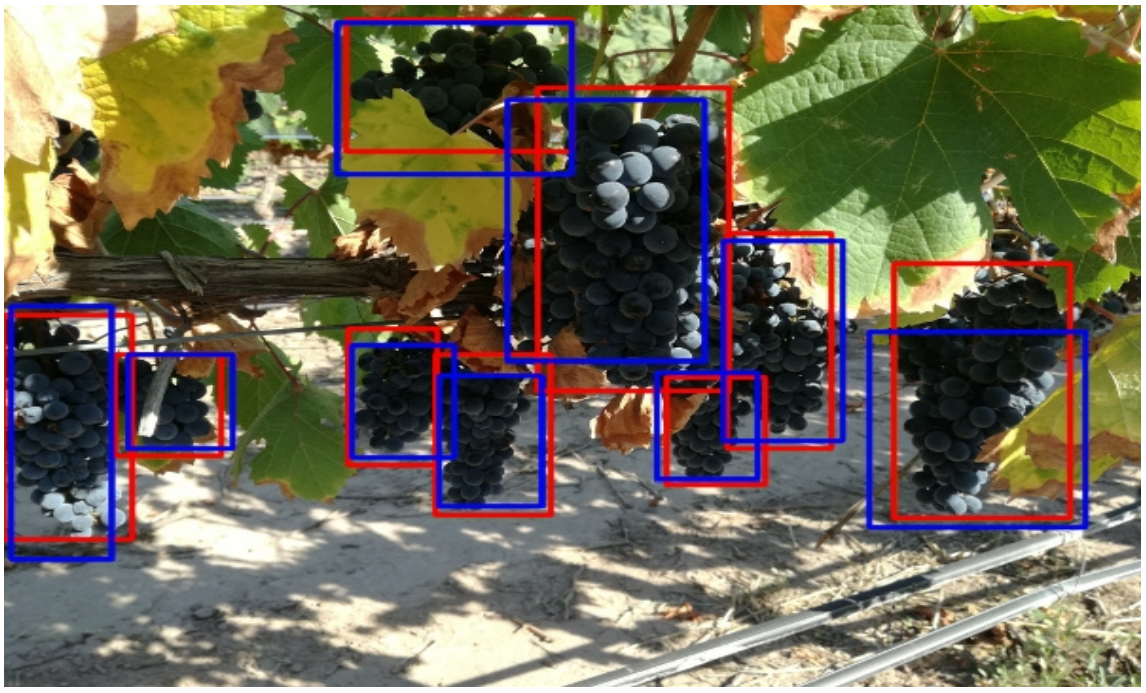


Figura 4.2. Ejemplo de todas detecciones satisfactorias en una imagen.

Por último,

5. Si un rectángulo detectado no se intersecta con ningún racimo etiquetado, entonces lo catalogué como OTRA DETECCIÓN. Este caso responde a muchas detecciones (que visualmente son correctas) de racimos de vista parcial que no fueron etiquetados inicialmente por ser muy pequeños o estar muy obstruidos, como se muestra en la Figura 4.5. Pero también a casos de detecciones incorrectas, como se muestra a la derecha de la Figura 4.6. Es decir, la categoría "otra detección" no necesariamente indica que la NN detectó algo incorrecto, sino que pudo haber detectado un racimo correctamente que nosotros no etiquetamos durante el proceso de anotación de imágenes.

Teniendo en cuenta el criterio anterior fue que hicimos las comparaciones que se detallan en el resto del Capítulo, salvo en la Sección 4.3.3.

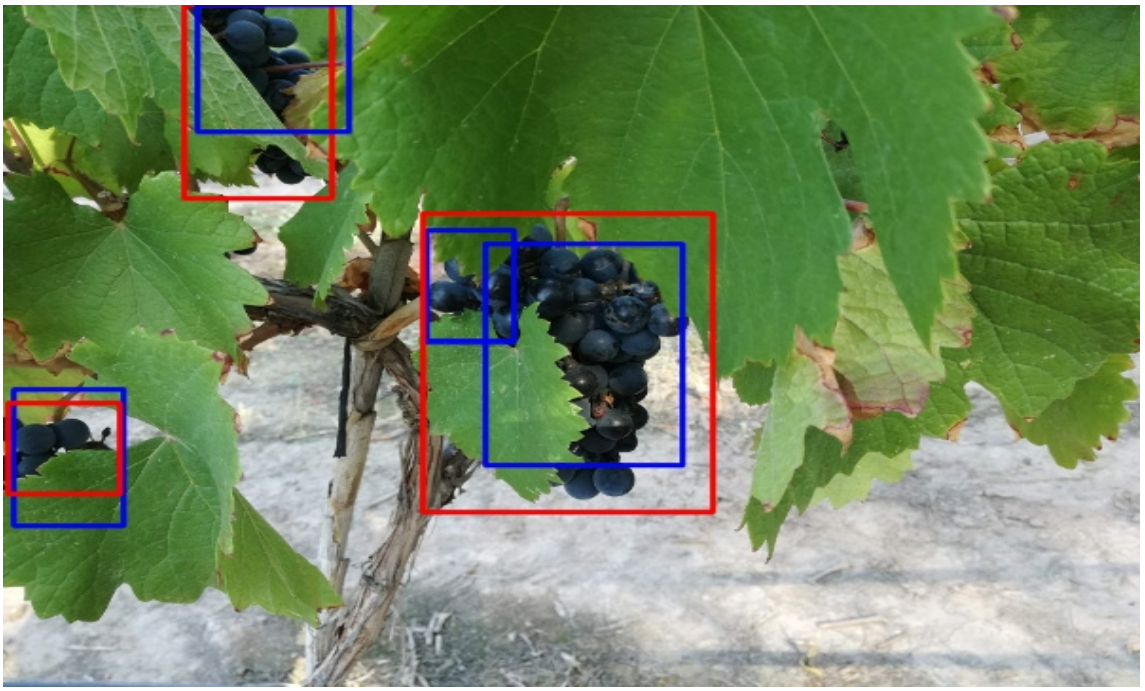


Figura 4.3. Ejemplo de un racimo etiquetado que es detectado como dos racimos más pequeños.

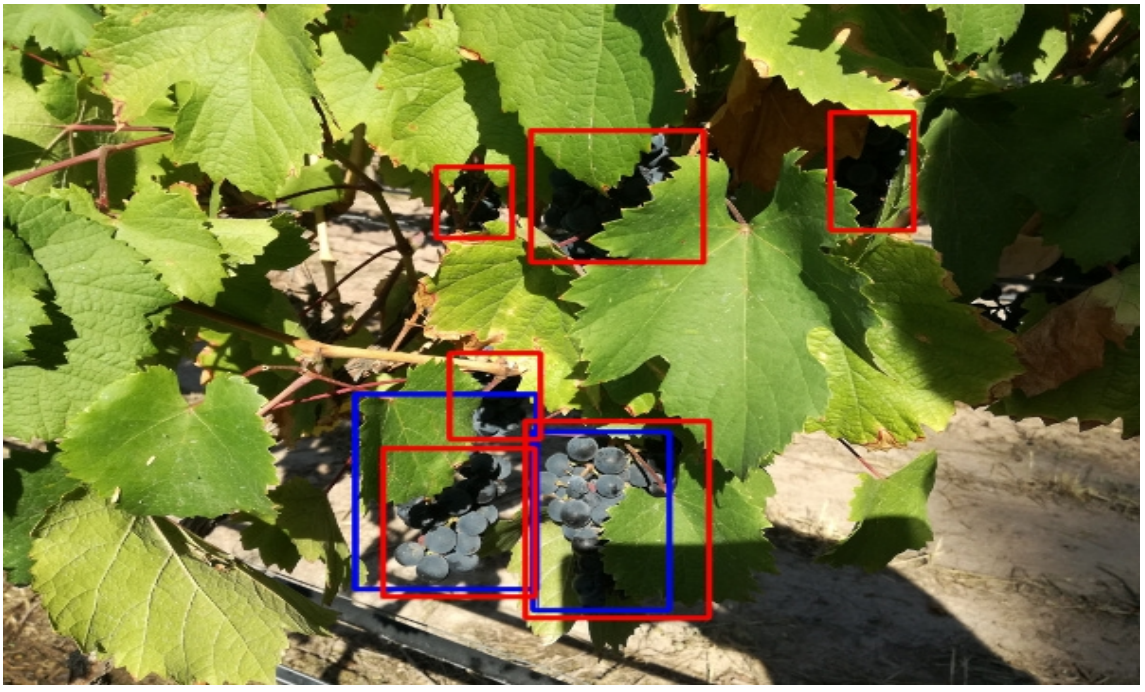


Figura 4.4. Ejemplo de racimos no detectados.

Figura 4.5. Ejemplo de "otras detecciones" correspondientes a racimos no etiquetados inicialmente.

Figura 4.6. Ejemplo de "otras detecciones" correspondientes a racimos no etiquetados inicialmente (a la izquierda del centro de la imagen) y de detecciones incorrectas (sobre el margen derecho).

4.3. Resultados de aplicar técnicas de aumento de datos

Cuando comenzamos a trabajar en la detección de racimos en este Seminario de Investigación, lo primero que conjeturamos fue que la red entrenada a partir de *bunch*600 lograría más y mejores detecciones que la entrenada a partir de *bunch*120, y nos preguntamos además si las detecciones obtenidas a partir del uso de técnicas de aumento de datos aplicadas a *bunch*120 igualarían los resultados obtenidos a partir de *bunch*600. Comenzamos entonces con estas pruebas usando tamaño de lote, `batch_size` = 2, tasa de aprendizaje, `LR` $= 1 \times 10^{-4}$, y 100 experimentos o épocas. Además iniciamos las pruebas usando el primero de los datasets de *bunch*600, es decir, *bunch*600.1.

Para visualizar el comportamiento del error a medida que se avanza con el entrenamiento, graficamos los valores `loss` y `val_loss` obtenidos luego de finalizar cada época en función del número de experimentos. En la Figura 4.7 observamos los errores de entrenamiento y validación para cada época para el dataset de *bunch*120 (izquierda), y para *bunch*600.1 (derecha), con los hiperparámetros antes mencionados.

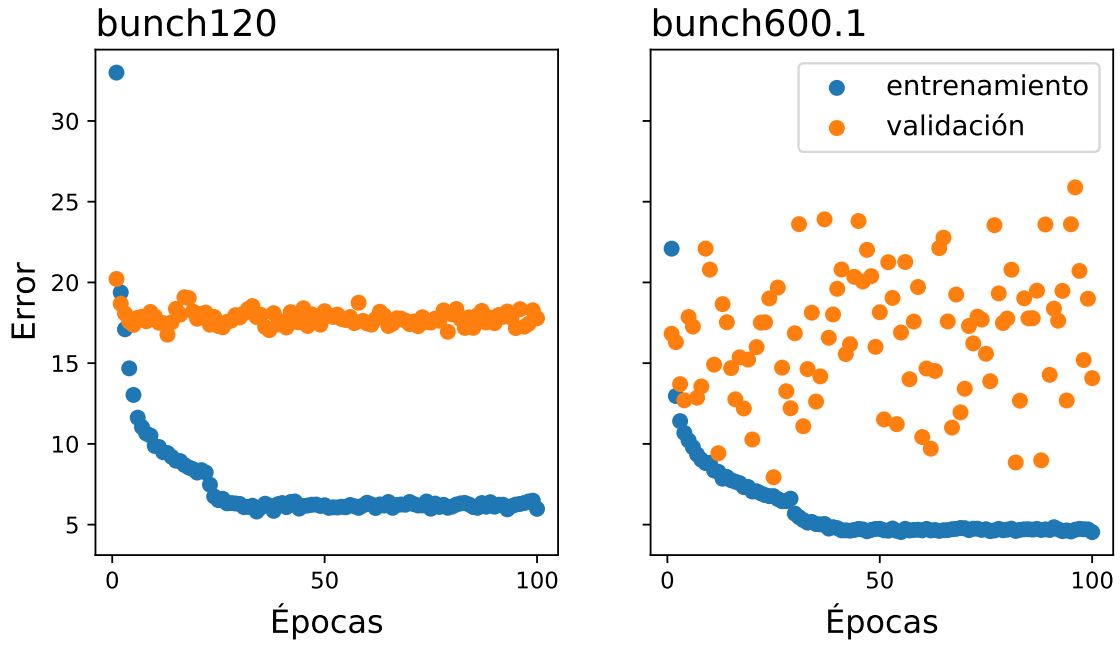


Figura 4.7. Errores de entrenamiento y validación de *bunch*120 y *bunch*600.1, entrenando con batch_size = 2 y LR $= 1 \times 10^{-4}$.

Como vemos, para *bunch*120 el error de entrenamiento disminuye a 6, mientras que el de validación oscila alrededor de 17, lo cual puede alertarnos de un sobreajuste, que es esperable dada la poca cantidad de datos presentes en el dataset. Por otro lado, llama mucho más la atención el comportamiento del error de validación para *bunch*600.1, que aparentemente no sigue ningún patrón. Para poder apreciarlo mejor, repetimos la gráfica uniendo ahora los puntos en la Figura 4.8 (izquierda), donde podemos ver que es extremadamente fluctuante. Esto nos llevó a cuestionarnos la separación de datos de *bunch*600.1

en $\mathcal{D}_{train}$ y $\mathcal{D}_{val}$, y notamos que por un error involuntario la mayoría de las fotografías de plano general de los espalderos habían terminado en $\mathcal{D}_{val}$, por lo que la red no contaba con suficientes ejemplos de entrenamiento de estas características. Para enmendar esto fue que armamos *bunch*600.2, eliminando los planos generales tomados a mayor distancia de los espalderos, por fotografías tomadas a menor distancia, e intercambiamos imágenes de $\mathcal{D}_{train}$ por imágenes de $\mathcal{D}_{val}$ a fin de evitar que el conjunto de validación contara solamente con un único tipo de fotografías. Así, repetimos el entrenamiento, evaluación de modelos y detección para *bunch*600.2 usando los mismos hiperparámetros, y graficamos nuevamente sus errores de entrenamiento y validación, los cuales se muestran en la Figura 4.8 (derecha). Para este conjunto de datos a la red le toma más épocas poder disminuir el error de entrenamiento, y además el error de validación mejora notablemente, por lo que decidimos tomar *bunch*600.2 como conjunto de referencia a la hora de comparar resultados con otras pruebas, y a partir de ahora nos referiremos a éste simplemente como *bunch*600.

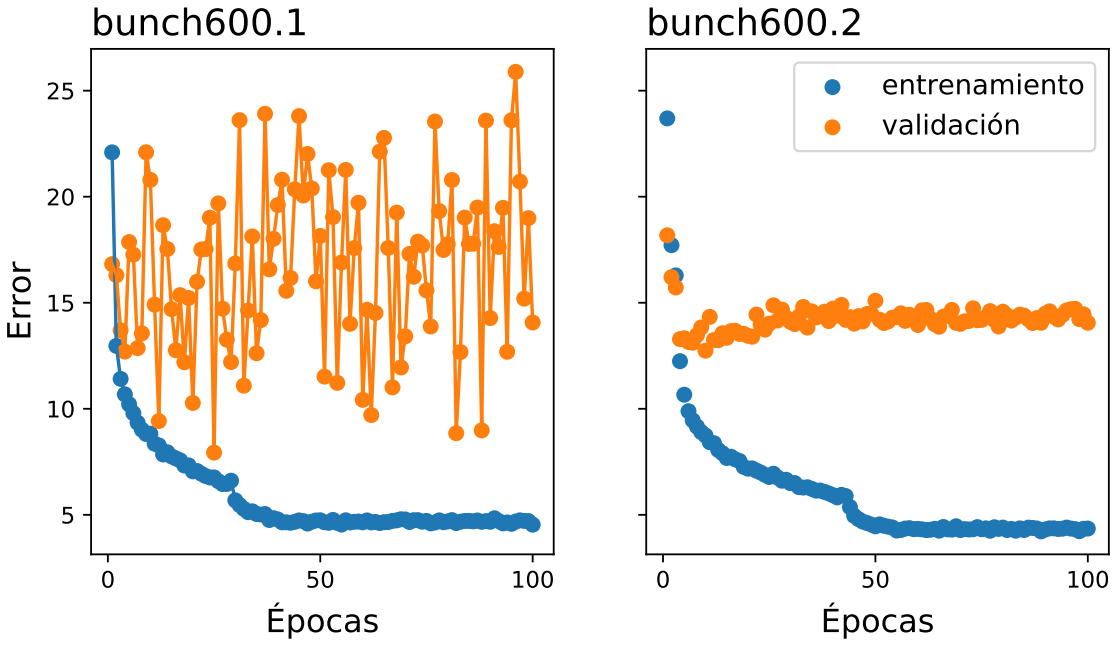


Figura 4.8. Errores de entrenamiento y validación de *bunch*600, entrenando con batch_size = 2 y LR = 1×10^{-4}.

Respecto a los modelos seleccionados, el valor de mAP para *bunch*120 fue de 0,589126, mientras que el de *bunch*600 fue de 0,682163, lo que es más de un 9 %. A continuación presentaremos los errores y mAP de las pruebas que duplicaron y quintuplicaron a *bunch*120.

4.3.1. Resultados de duplicar *bunch*120

Como vimos en la Sección 3.2 hicimos pruebas duplicando y quintuplicando el conjunto de datos de entrenamiento y validación de *bunch*120. Podemos observar los errores para las pruebas que duplicaron el dataset en la Figura 4.9.

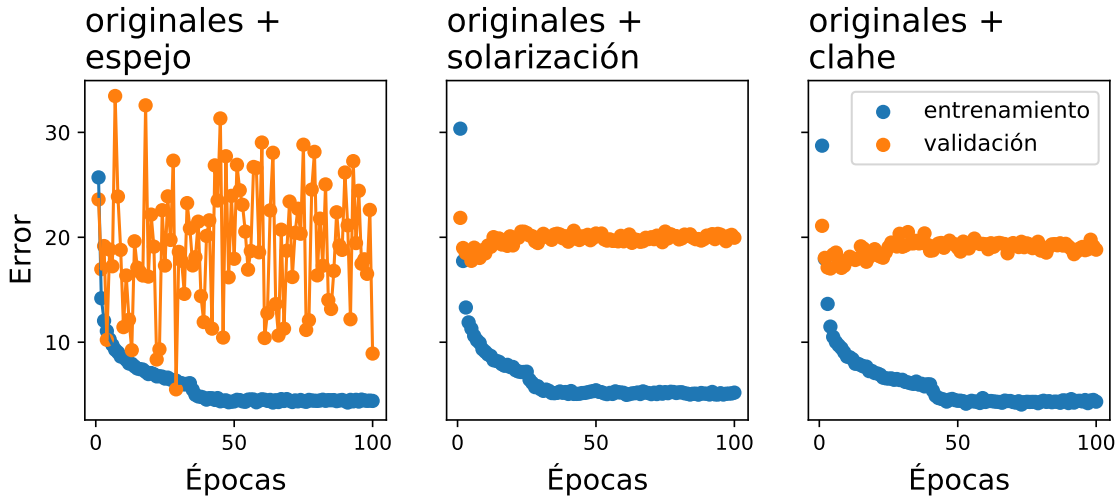


Figura 4.9. Errores de entrenamiento y validación de conjuntos que duplican *bunch*120, entrenando con batch_size = 2 y LR = 1×10^{-4}.

Una vez más observamos que el error de validación tiene un comportamiento fluctuante para el dataset formado a partir de las imágenes originales de *bunch*120 y la reflexión o espejado de cada una de ellas (Figura 4.9 izquierda). Sin embargo, para los casos donde se aplicó solarización (Figura 4.9 medio) o CLAHE (Figura 4.9 derecha), los errores de validación muestran un pequeño incremento y luego se mantienen oscilantes alrededor de un valor. Para el primer caso, el motivo de la fluctuación podría ser una mala elección de los hiperparámetros `batch_size` y `LR`, entre otras razones, pero dado que la idea es comparar resultados solamente modificando los conjuntos de entrenamiento y validación, y dado que no tuvimos mucho tiempo para hacer más entrenamientos, es que no pudimos hacer una repetición de esta prueba con otros valores de hiperparámetros. Por otro lado, el último caso es el que tiene un mejor comportamiento, ya que a pesar de incrementarse el error de validación, hacia el final muestra una tendencia a disminuir nuevamente.

Respecto a los valores mAP de los modelos seleccionados, tenemos que originales + solarización alcanzó un 0,542306, originales + espejo un 0,571344, y originales + CLAHE un 0,593465, llegando este último a un mAP superior a la prueba de *bunch*120 con solo imágenes originales.

4.3.2. Resultados de quintuplicar *bunch*120

Al igual que en la Sección anterior, presentamos los errores así como los valores de mAP para las pruebas que quintuplicaron el dataset *bunch*120. En primer lugar, la Figura 4.10 muestra los errores de entrenamiento y validación para los entrenamientos con cuatro valores de atenuación de un mismo ruido, ya sea el Gaussiano (izquierda), el Laplaciano (medio) o el de Poisson (derecha), mientras que la Figura 4.11 muestra los errores de entrenar con cuatro valores de sigma para empañar (izquierda) o cuatro radios distintos

para difuminar como una pintura (derecha).

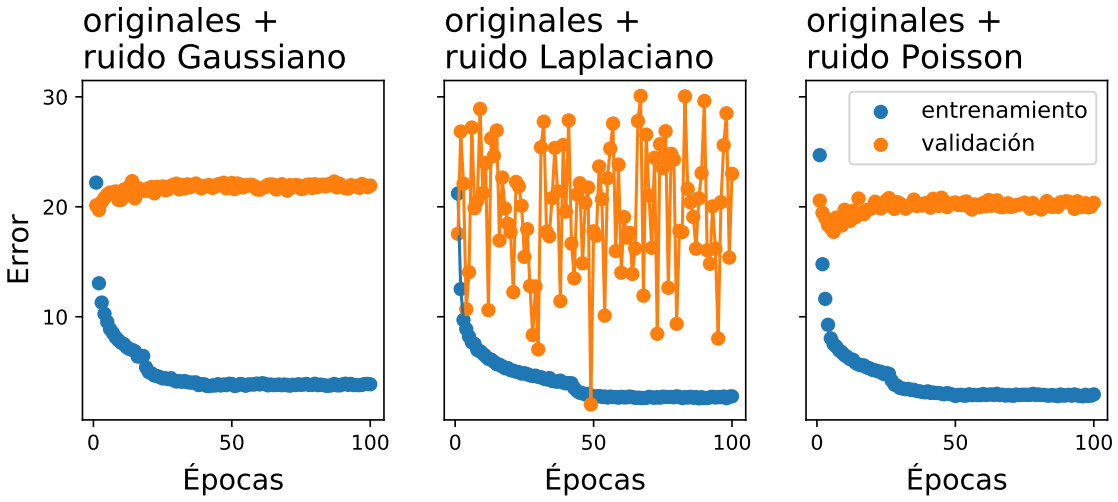


Figura 4.10. Errores de entrenamiento y validación de conjuntos que quintuplican *bunch*120 (ruidos), entrenando con batch_size = 2 y LR = 1×10^{-4}.

Nuevamente observamos casos donde el error de validación fluctúa bruscamente y casos donde tiene un comportamiento más estable, pero donde no logra disminuir. En estos resultados coincide además que al error de entrenamiento le toma más de 40 épocas poder estabilizarse cuando hay fluctuación en el error de validación, mientras que unas 30 épocas para las otras situaciones. Al igual que con la prueba de originales + reflexión horizontal, no pudimos volver a probar originales + ruido Laplaciano ni originales + pintura con otros valores de hiperparámetros, por cuestiones de tiempo.

Por otro lado, los valores mAP alcanzados por los modelos seleccionados del entrenamiento con ruidos fueron de: 0,366291 para originales + ruido Gaussiano, 0,473912 para originales + ruido Laplaciano, 0,559054 para originales + ruido de Poisson, mientras que originales + empañado tuvo un 0,561958 y originales + pintura un 0,6010, siendo este último el único que superó el valor de mAP del modelo entrenado con las imágenes originales de *bunch*120. Ningún modelo entrenado usando técnicas de aumento de datos, ya sea las que duplicaron como las que quintuplicaron la base de datos, logró igualar o superar la medida de rendimiento del modelo entrenado con *bunch*600, sin embargo veremos que el aplicar algunas de estas técnicas lleva a un incremento en la exhaustividad del modelo de detección, lo cual es deseable en los casos en los que se tiene un conjunto de entrenamiento escaso como lo es *bunch*120.

Dado que los valores de mAP de originales + ruido Gaussiano y originales + ruido Laplaciano son menores a 0,5, no los tendremos en cuenta a la hora de estimar la exhaustividad o la precisión de las detecciones.

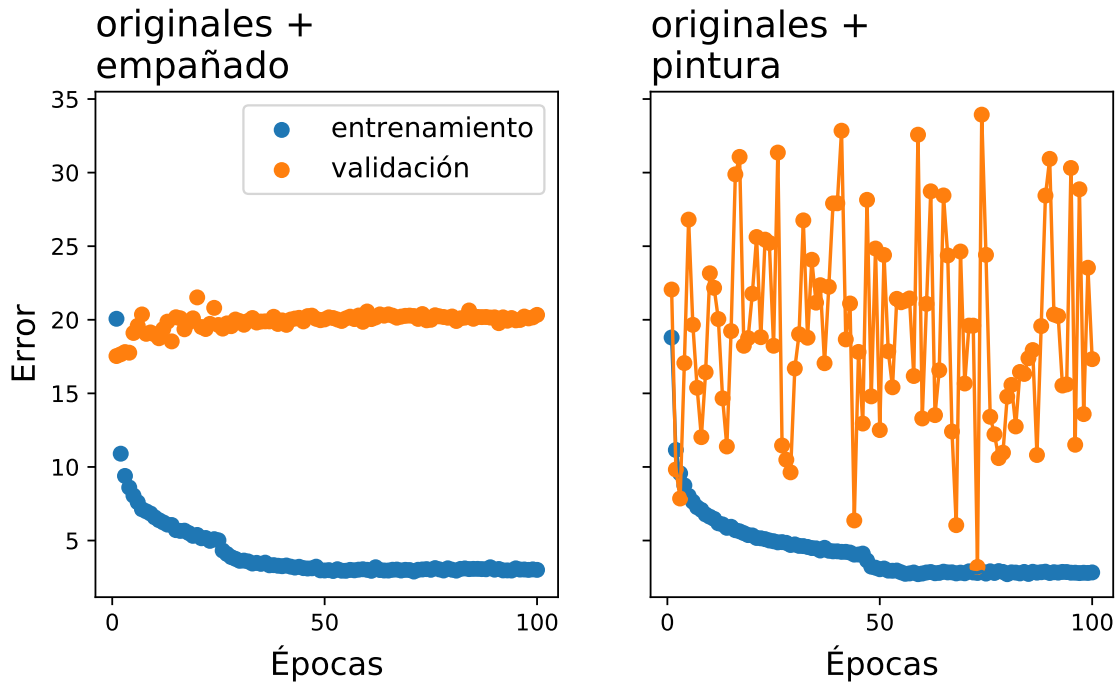


Figura 4.11. Errores de entrenamiento y validación de conjuntos que quintuplican *bunch*120 (empañado y pintura), entrenando con batch_size $= 2$ y LR $= 1 \times 10^{-4}$.

4.3.3. Comparación de exhaustividad y precisión

Podemos resumir el criterio explicado en la Sección 4.2 como:

1. Son "posibles detecciones" de un racimo etiquetado aquellos rectángulos que se intersecten en al menos un 25 % con el rectángulo etiquetado.

2. Si para un racimo etiquetado sólo hay una posible detección, se considera "detectado correctamente" si ambos rectángulos se intersectan en al menos un 60 %.

3. Si para un racimo etiquetado existe más de una posible detección, se considera "detectado correctamente" si la suma de las intersecciones de cada detección con el rectángulo etiquetado es de al menos el 60 % de éste.

4. Si para un racimo etiquetado no existen posibles detecciones, o si éstas no satisfacen 2 ó 3, entonces se considera "no detectado".

5. Si un rectángulo detectado no se intersecta con ningún racimo etiquetado, entonces se considera "otra detección".

Teniendo en cuenta lo anterior, el Cuadro 4.1 muestra la diferencia de detecciones entre *bunch*120 y *bunch*600, los cuales tienen 100 y 500 imágenes de entrenamiento, respectivamente. Recordemos que el nombre "originales" hace referencia a que las imágenes no tienen

ningún tipo de filtro o transformación aplicada. Este Cuadro, así como los demás de esta Sección, resume la siguiente información:

Racimos detectados correctamente: cantidad de racimos etiquetados que fueron detectados correctamente. Las detecciones satisfacen 1 y 2 ó 1 y 3.

Racimos no detectados: cantidad de racimos que no fueron detectados. En estos casos, ninguna de las detecciones satisface 1 para el racimo etiquetado en cuestión, ó si lo hace, entonces no satisface 2 ó 3.

Exhaustividad: porcentaje de racimos etiquetados que fueron detectados correctamente, es decir, exhaustividad = (Racimos detectados correctamente) / 641 $\times$ 100.

Detecciones del modelo: cantidad de detecciones predichas por el modelo.

Detecciones racimos satisfactorias: aquellas detecciones predichas que satisfacen 1 y 2 ó 1 y 3.

Detecciones racimos no satisfactorias: aquellas detecciones predichas que satisfacen 1, pero no satisfacen 2 ó 3.

Otras detecciones: aquellas detecciones que no satisfacen 1, que podrían – o no – corresponderse con racimos de vista parcial, que no fueron etiquetados inicialmente.

Precisión: exactitud de los modelos para detectar racimos, esto es,
precisión = (Detecciones racimos satisfactorias) / (Detecciones del modelo) $\times$ 100.

Promedio certezas: valor promedio del nivel de certeza de las detecciones de racimos, hayan sido o no satisfactorias.

Desviación estándar certezas: desviación estándar del nivel de certeza de las detecciones de racimos, hayan sido o no satisfactorias.

mAP: valor mAP reportado por `ImageAI`.

De manera similar, el Cuadro 4.2 muestra las detecciones alcanzadas por la combinación entre *bunch*120 (imágenes originales) y la aplicación de un tipo de transformación, que aumente a 200 el número de imágenes de entrenamiento, mientras que el Cuadro 4.3 muestra las detecciones logradas con las combinaciones de las imágenes originales y cuatro valores de algún tipo de filtro, llegando a 500 imágenes de entrenamiento por prueba. En ambos hemos agregado los resultados del Cuadro 4.1 para facilitar su comparación.

Si bien varias detecciones de racimos no son satisfactorias, quisimos analizar qué rango de certeza tienen TODAS las detecciones de la red, independientemente de si se tratan de detecciones que llevan a que un racimo etiquetado sea considerado "detectado correctamente", o no. Por eso solo distinguimos entre las detecciones que se corresponden con

Cuadro 4.1. Detecciones de *bunch*120 y *bunch*600, entrenando con batch_size = 2 y LR = 1×10^{-4}, durante 100 épocas.

	*bunch*120 originales	*bunch*600 originales
Racimos detectados correctamente	455	568
Racimos no detectados	186	73
Exhaustividad	**70,98** %	**88,61** %
Detecciones del modelo	613	670
Detecciones racimos satisfactorias	427	551
Detecciones racimos no satisfactorias	61	19
Otras detecciones	125	100
Precisión	**69,66** %	**82,24** %
Promedio certezas	50,31	54,18
Desviación estándar certezas	9,73	9,36
mAP	**0,589126**	**0,682163**

racimos etiquetados (ya sean o no detecciones satisfactorias), de las "otras detecciones". En la Figura 4.12 se pueden ver los porcentajes de certeza de las detecciones obtenidas luego del entrenamiento con imágenes de *bunch*120 y *bunch*600, sin la aplicación de transformaciones o filtros. A la izquierda, los gráficos (a) y (c) tienen en cuenta las certezas de las detecciones correspondientes a racimos etiquetados, mientras que los gráficos (b) y (d) de la derecha tienen en cuenta las otras detecciones.

En forma análoga, la Figura 4.13 tiene a su izquierda las detecciones correspondientes a racimos etiquetados, obtenidas de los entrenamientos realizados con las imágenes originales de *bunch*120 junto con la ecualización CLAHE (a), junto a la transformación de reflexión (c), y junto al filtro de solarización (e). En cambio a su derecha se muestran las otras detecciones. De igual modo la Figura 4.14 muestra los gráficos correspondientes al entrenamiento con imágenes con ruido de Poisson, empañado y pintura.

Como es esperable, la NN logra detectar correctamente un 88 % de los racimos etiquetados a partir del entrenamiento con el dataset *bunch*600 y este modelo tiene una precisión del 82 %, mientras que las detecciones correctas a partir del entrenamiento con *bunch*120 sólo llegan al 70 % y la precisión en este caso es de 69 %. Sin embargo aplicar algunas técnicas de aumento de datos llevan a incrementar el número de detecciones. Al duplicar el dataset original usando filtros como CLAHE, espejado o solarización se logra un aumento de entre el 4 y el 13 % de detecciones correctas, además que la precisión es superior al modelo entrenado solamente con las imágenes originales de *bunch*120. Sin embargo, la precisión del modelo entrenado junto con solarización es menor a la del modelo entrenado con imágenes originales y espejadas. Por otro lado, quintuplicar el dataset original sólo

Cuadro 4.2. Detecciones datasets que duplican el conjunto de entrenamiento, entrenando con batch_size $= 2$ y LR $= 1 \times 10^{-4}$, durante 100 épocas.

	*bunch*120 originales	originales + clahe	originales + espejadas	originales + solarización	*bunch*600 originales
Racimos detectados correctamente	455	458	480	536	568
Racimos no detectados	186	183	161	105	73
Exhaustividad	**70,98 %**	**71,45 %**	**74,88 %**	**83,62 %**	**88,61 %**
Detecciones del modelo	613	524	646	805	670
Detecciones racimos satisfactorias	427	418	475	565	551
Detecciones racimos no satisfactorias	61	26	48	23	19
Otras detecciones	125	80	123	217	100
Precisión	**69,66 %**	**79,77 %**	**73,53 %**	**70,19 %**	**82,24 %**
Promedio certezas	50,31	51,39	55,93	54,46	54,18
Desviación estándar certezas	9,73	11,39	11,25	13,82	9,36
mAP	**0,589126**	**0,593465**	**0,571344**	**0,542306**	**0,682163**

representa un 2 % más de detecciones correctas cuando se usa el filtro de pintura, pero su precisión es superior a las antes nombradas.

Es notable obtener un porcentaje de detección mayor cuando se entrena con las imágenes originales + espejadas, que cuando se entrena con las originales + Poisson, u originales + empañado u originales + pintura. Es decir, con sólo duplicar los datos usando esta transformación particular se llega a superar la exhaustividad obtenida de quintuplicar los datos, siendo que la primera opción reduce significativamente el tiempo de cómputo. Sin embargo no hay que perder de vista que originales + reflexión horizontal cuenta con mayores fluctuaciones en el error de validación (Figura 4.9 izquierda), que las otras pruebas que duplican el dataset original. Por lo que podríamos esperar que para otros valores de hiperparámetros, la aplicación de esta transformación tenga un valor de exhaustividad más alto. Por otro lado, si bien el modelo generado del entrenamiento con imágenes originales + solarización es el que detecta correctamente mayor número de racimos etiquetados, es también el que tiene precisión más baja, lo que lo hace menos fiable, de hecho es el que detecta mayor número de "otras detecciones".

A su vez, los modelos generados a partir de los entrenamientos aplicando el filtro solarización, de reflexión horizontal y de ruido de Poisson son los que detectan mayor número de racimos con niveles de certeza altos ($> 50\,\%$). Por otro lado, los modelos entrenados con CLAHE o difuminado por pintura son los únicos que tienen detecciones con nivel de confianza mayor a 87 %, superando incluso a los detectados con *bunch*600.

Por otro lado, vemos que los modelos entrenados con las imágenes originales de *bunch*120, con originales + solarización y originales + CLAHE son los que tienen mayor cantidad de otras detecciones con niveles de certeza bajos ($< 50\,\%$). Sin embargo este último, junto con el modelo entrenado con originales + pintura tiene otras detecciones con niveles de certeza superiores al 80 %, lo cual llama mucho la atención.

Cuadro 4.3. Detecciones datasets que quintuplican el conjunto de entrenamiento, entrenando con batch_size $= 2$ y LR $= 1 \times 10^{-4}$, durante 100 épocas.

	*bunch*120 originales	originales + ruido Poisson	originales + empañado	originales + pintura	*bunch*600 originales
Racimos detectados correctamente	455	452	453	467	568
Racimos no detectados	186	189	188	174	73
Exhaustividad	**70,98** %	**70,51** %	**70,67** %	**72,85** %	**88,61** %
Detecciones del modelo	613	599	525	537	670
Detecciones racimos satisfactorias	427	452	430	426	551
Detecciones racimos no satisfactorias	61	40	37	25	19
Otras detecciones	125	107	58	86	100
Precisión	**69,66** %	**75,46** %	**81,9** %	**79,33** %	**82,24** %
Promedio certezas	50,31	58,63	54,79	48,07	54,18
Desviación estándar certezas	9,73	9,77	9,36	13,29	9,36
mAP	**0,589126**	**0,559054**	**0,561958**	**0,6010**	**0,682163**

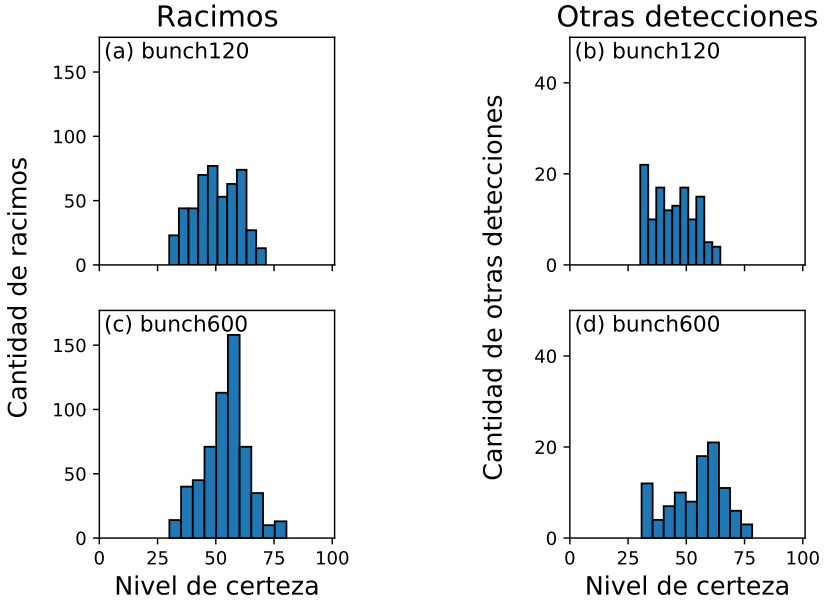


Figura 4.12. Porcentajes de certeza de las detecciones obtenidas luego del entrenamiento con las imágenes originales de *bunch*120 (arriba) y *bunch*600 (abajo).

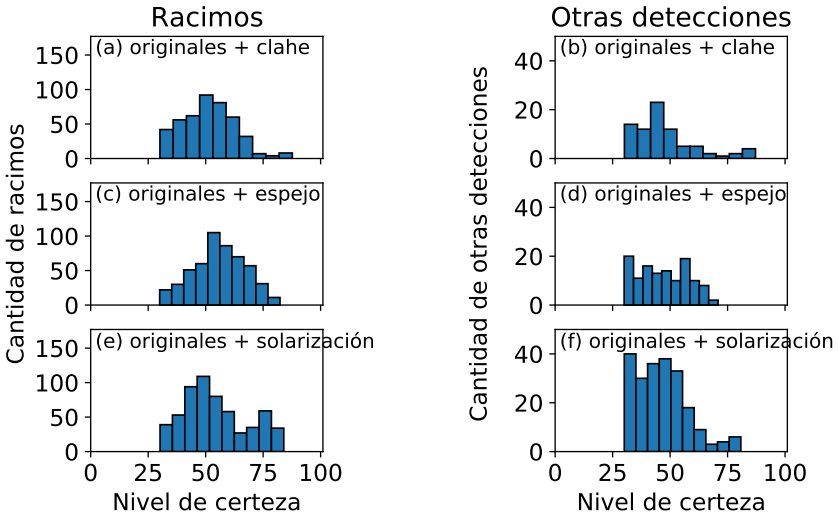


Figura 4.13. Porcentajes de certeza de las detecciones obtenidas luego del entrenamiento con las imágenes originales de *bunch*120 junto a la ecualización CLAHE (arriba), junto a sus reflexiones (medio), y junto a las imágenes con solarización (abajo).

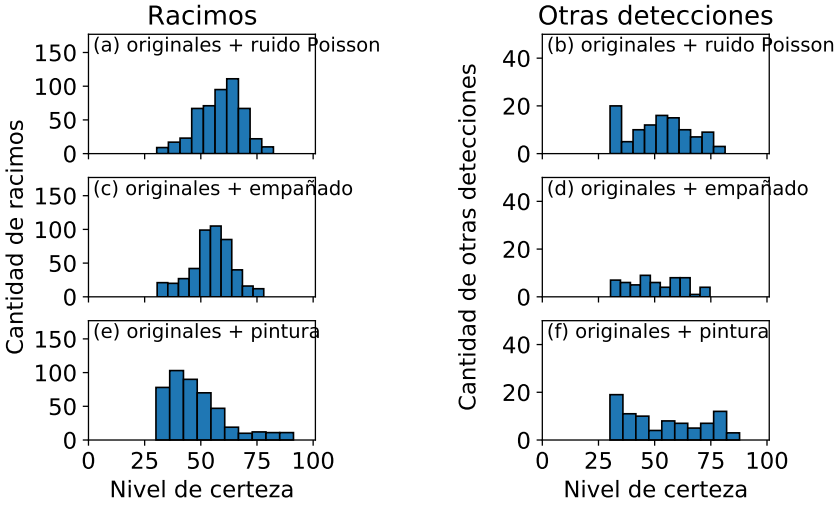


Figura 4.14. Porcentajes de certeza de las detecciones obtenidas luego del entrenamiento con las imágenes originales de *bunch*120 junto a cuatro valores de Poisson (arriba), de empañado (medio) y de pintura (abajo).

Exhaustividad y precisión al cambiar el criterio

Para cerrar esta sección de pruebas de técnicas de aumento de datos, presentamos los resultados de exhaustividad y precisión para los mismos modelos generados del entrenamiento con imágenes originales de *bunch*120, imágenes originales de *bunch*600 e imágenes originales + espejadas, salvo que cambiamos ahora el criterio para determinar si un racimo etiquetado fue detectado correctamente. En este caso, tomamos el vaor IoU en lugar de usar solamente intersecciones (IoU de 0,25 en la condición 1 e IoU de 0,6 en las condiciones 2 y 3). Los resultados se muestran en el Cuadro 4.4.

Cuadro 4.4. Exhaustividad y precisión al evaluar IoU en lugar de sólo intersecciones.

	*bunch*120 originales	*bunch*600 originales	originales + espejadas
Racimos detectados correctamente	298	420	370
Racimos no detectados	343	221	271
Exhaustividad	**46**,**49** %	**65**,**52** %	**57**,**72** %
Detecciones del modelo	613	670	646
Detecciones racimos satisfactorias	313	437	388
Detecciones racimos no satisfactorias	172	130	129
Otras detecciones	128	103	129
Precisión	**51**,**06** %	**65**,**22** %	**60**,**06** %

Notamos que a pesar de disminuir la exhaustividad y precisión en todos los casos, seguimos observando el mismo comportamiento que antes, esto es, el modelo entrenado a partir de *bunch*600 detecta mayor cantidad de racimos con mayor precisión, mientras que aplicar técnicas de aumento de datos logra incrementar el número de racimos detectados en comparación a lo obtenido del entrenamiento con únicamente imágenes originales de *bunch*120.

Una vez que completamos todas las pruebas descriptas hasta ahora y las comparamos entre sí notamos que existía evidencia de sobreajuste, como pudimos observar en las gráficas de error de entrenamiento y validación. Por esto nos preguntamos si podría solucionarse este problema modificando los valores de los hiperparámetros tamaño de lote (`batch_size`) y tasa de aprendizaje (`LR`).

4.4. Resultados de probar diferentes hiperparámetros

Para probar el comportamiento de la NN con diferentes hiperparámetros, elegimos usar las imágenes originales de *bunch*120 como conjunto de entrenamiento, y entrenar

durante 50 épocas, a fin de disminuir el tiempo de cómputo. Comenzamos modificando un solo hiperparámetro por vez, y comparamos el valor de exhaustividad alcanzado con cada modelo, calculado de acuerdo a lo establecido en la Sección 4.2, así como el valor mAP.

4.4.1. Tamaños de lote distintos

Al comenzar estas pruebas sólo modificamos el valor `batch_size`: en lugar de fijarlo en 2, usamos los valores 4, 8 y 12, manteniendo en todos los casos una tasa de aprendizaje `LR` $= 1 \times 10^{-4}$. Los resultados se muestran en el Cuadro 4.5, y las gráficas de error en la Figura 4.15. Vale aclarar que intentamos aumentar a 16 y 32 el tamaño de lote, sin embargo esas pruebas no lograron completarse porque las GPUs utilizadas no cuentan con suficiente memoria.

Cuadro 4.5. Exhaustividad y mAP de modelos al entrenar con imágenes originales de *bunch*120 durante 50 épocas, usando distintos tamaños de lote.

	`batch_size` = 2	`batch_size` = 4	`batch_size` = 8	`batch_size` = 12
Racimos detectados correctamente	450	466	491	519
Racimos no detectados	191	175	150	122
Exhaustividad	**70,22 %**	**72,7 %**	**76,6 %**	**80,97 %**
Otras detecciones	107	98	85	288
mAP	**0,601557**	**0,654041**	**0,605176**	**0,659970**
Tiempo entrenamiento (s)	22286,24	13163,46	9522,48	8987,77

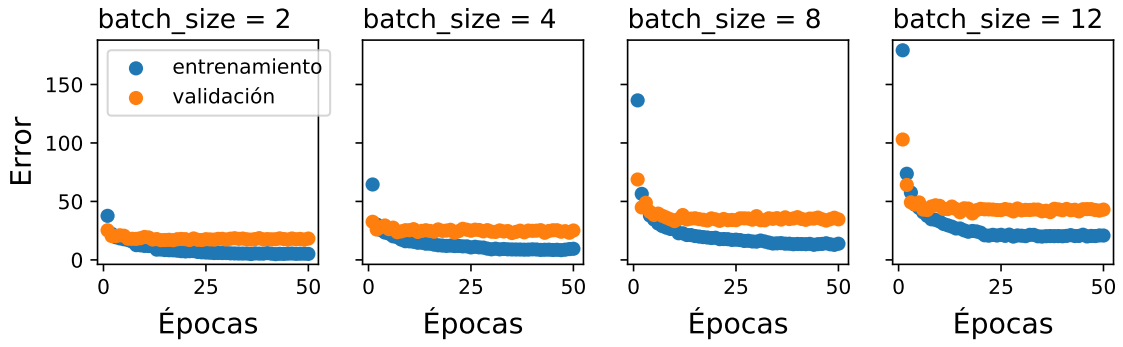


Figura 4.15. Errores de entrenamiento y validación para distintos tamaños de lote.

Teniendo en cuenta la exhaustividad, el valor mAP y las gráficas de los errores, vemos que los mejores tamaños de lote son de 4 y 8. Basándonos además en la cantidad de otras detecciones y el tiempo de entrenamiento, concluimos que el tamaño más adecuado es de 8. Así, en las siguientes pruebas utilizamos este tamaño de lote y sólo modificamos la tasa de aprendizaje.

4.4.2. Tasas de aprendizaje distintas

Como habrán notado en la salida impresa mostrada en el Código 4.1, la tasa de aprendizaje no es un hiperparámetro a modificar cuando se usa `ImageAI`, y su valor por defecto es 1×10^{-4}, como se ha usado hasta ahora. Por lo tanto, para poder usar un valor distinto, busqué en el código fuente de `ImageAI` para cambiar la línea donde fija dicho valor, el código puede observarse en GitHub[1]. Hicimos dos pruebas, una reduciéndolo a la mitad y otra aumentándolo en 1,5 veces, nuevamente entrenando con las imágenes originales de *bunch*120, durante 50 épocas, usando en este caso un tamaño de lote de 8. Los resultados se muestran en el Cuadro 4.6, y las gráficas de error en la Figura 4.17.

Cuadro 4.6. Exhaustividad y mAP de modelos al entrenar con imágenes originales de *bunch*120 durante 50 épocas, usando un tamaño de lote de 8, y distintas tasas de aprendizaje.

	LR $= 1 \times 10^{-4}$	LR $= 5 \times 10^{-5}$	LR $= 1.5 \times 10^{-4}$
Racimos detectados correctamente	491	500	505
Racimos no detectados	150	141	136
Exhaustividad	**76,6 %**	**78 %**	**78,78 %**
Otras detecciones	85	36	85
mAP	**0,605176**	**0,622603**	**0,634476**

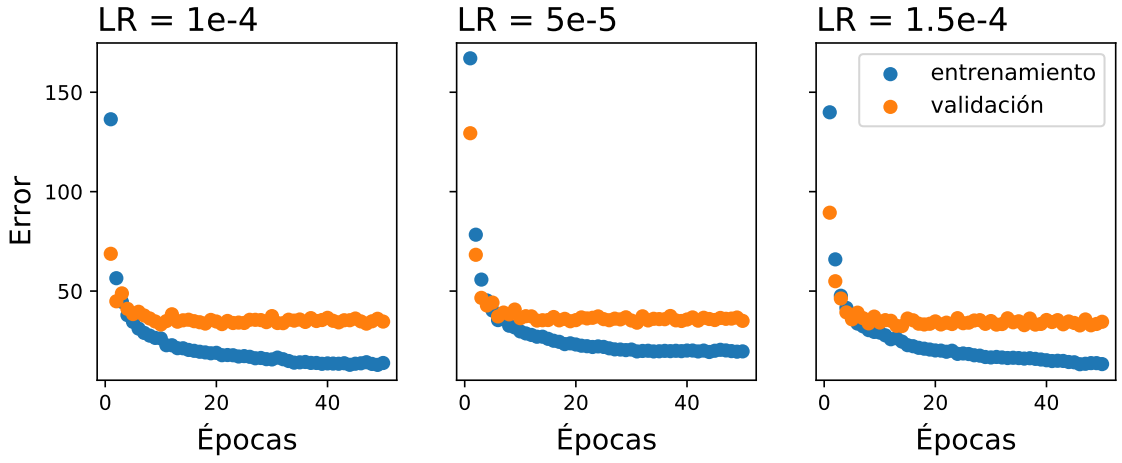


Figura 4.16. Errores de entrenamiento y validación para distintas tasas de aprendizaje.

Dado que no existe una diferencia significativa entre las distintas pruebas, decidimos continuar entrenando con el valor de tasa de aprendizaje que viene por defecto, es decir, en las pruebas que siguen se usó LR $= 1 \times 10^{-4}$.

[1] `https://github.com/OlafenwaMoses/ImageAI/blob/master/imageai/Detection/Custom/__init__.py`

Habiendo determinado valores adecuados de tamaño de lote y tasa de aprendizaje, procedimos a repetir la prueba base *bunch*120 vs. *bunch*600 con estos valores de hiperparámetros.

4.4.3. Puesta a prueba de los hiperparámetros elegidos

En el Cuadro 4.7 vemos la comparación de los nuevos modelos surgidos del entrenamiento con los mismos conjuntos de datos que comparamos en el Cuadro 4.1, salvo que ahora usamos nuevos valores de hiperparámetos.

Cuadro 4.7. Detecciones de *bunch*120 y *bunch*600, entrenando con batch_size $= 8$ y LR $= 1 \times 10^{-4}$, durante 100 épocas.

	*bunch*120 originales	*bunch*600 originales
Racimos detectados correctamente	482	616
Racimos no detectados	159	25
Exhaustividad	**75,2** %	**96,1** %
Detecciones del modelo	565	855
Detecciones racimos satisfactorias	452	617
Detecciones racimos no satisfactorias	38	4
Otras detecciones	75	234
Precisión	**80,0** %	**72,16** %
Promedio certezas	62,45	59,92
Desviación estándar certezas	14,74	11,43
mAP	**0,628331**	**0,750801**

Podemos observar que tanto la exhaustividad como el valor de mAP aumentan luego del entrenamiento con nuevos valores de hiperparámetros, e incluso el valor promedio del nivel de certeza también es superior. La precisión del modelo entrenado a partir de *bunch*120 alcanza un 80 %, sin embargo la precisión del modelo entrenado a partir de *bunch*600 disminuye. Vemos también que este último eleva a 234 el número de otras detecciones, aunque por otro lado son solamente 4 las detecciones no satisfactorias de racimos etiquetados. Por otro lado, en la Figura 4.7 vemos que los errores de entrenamiento y validación son superiores a los obtenidos del entrenamiento con tamaño de lote 2 (Figuras 4.12 y 4.8). Además el error de validación para *bunch*120 permanece constante, mientras que el de *bunch*600 tiene un pequeño aumento antes de estabilizarse, el cual es menor al observado para el caso de tamaño de lote 2.

En este punto quisimos repetir las pruebas discutidas en la Sección 4.3 para los nuevos valores de hiperparámetros, sin embargo por cuestiones de tiempo y disponibilidad del

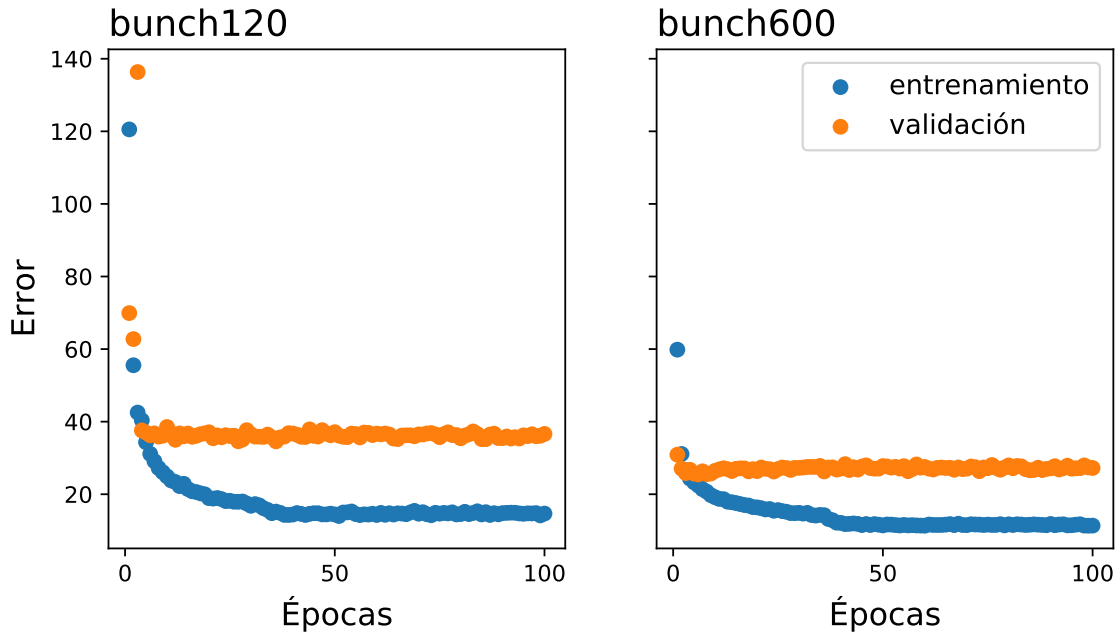


Figura 4.17. Errores de entrenamiento y validación de *bunch*120 y *bunch*600, entrenando con batch_size $= 8$ y LR $= 1 \times 10^{-4}$.

hardware esto no fue posible.

A continuación presentamos en el Cuadro 4.8 un resumen de los valores mAP, exhaustividad y precisión de las pruebas que comparan los modelos generados a partir del entrenamiento con *bunch*120 (y las técnicas de aumento de datos) y con *bunch*600, para tamaños de lote 2 y 8, de acuerdo al criterio de la Sección 4.2.

Cuadro 4.8. Resumen valores mAP, exhaustividad y precisión de las pruebas realizadas.

	batch_size = 2								batch_size = 8	
	Originales		Duplican			Quintuplican[a]			Originales	
	*bunch*120	*bunch*600	originales + clahe	originales + espejadas	originales + solarización	originales + ruido Poisson	originales + empañado	originales + pintura	*bunch*120	*bunch*600
Exhaustividad	70,98 %	88,61 %	71,45 %	74,88 %	83,62 %	70,51 %	70,67 %	72,85 %	75,2 %	96,1 %
Precisión	69,99 %	82,24 %	79,77 %	73,53 %	70,19 %	75,46 %	81,9 %	79,33 %	80,0 %	72,16 %
mAP	0,5891	0,6821	0,5935	0,5714	0,5423	0,5591	0,562	0,6010	0,6283	0,7508

[a] Las pruebas *originales + ruido Gaussiano* y *originales + ruido Laplaciano* obtuvieron un mAP de 0,3663 y 0,4739, respectivamente.

No forman parte del Cuadro porque no calculamos sus valores de exhaustividad y precisión por ser modelos con un valor mAP menor a 0,5.

4.5. Otros interrogantes

A medida que avanzamos con la investigación nos preguntamos si el desempeño de los modelos dependería del orden en que la red accede a las imágenes en la etapa de entrenamiento, así como qué tanto cambiarían las medidas de rendimiento al usar distintos conjuntos de entrenamiento y validación de 100 y 20 imágenes respectivamente.

Por ello, tomando los resultados del entrenamiento con *bunch*120 durante 100 épocas con un tamaño de lote de 8 y tasa de aprendizaje de 1×10^{-4} como punto de referencia (Cuadro 4.7), y llamándola "Prueba 1", hicimos cuatro nuevas pruebas manteniendo el conjunto de validación fijo y tomando de forma aleatoria las imágenes de entrenamiento, y cuatro pruebas juntando las fotografías de entrenamiento y validación y tomando de forma aleatoria las imágenes para formar nuevos conjuntos para entrenar y validar de 100 y 20 fotos cada uno. En todas estas pruebas usamos los mismos hiperparámetros ya mencionados.

4.5.1. Conjunto de entrenamiento aleatorio, conjunto de validación fijo

En el Cuadro 4.9 se observan los valores de exhaustividad y mAP de los modelos seleccionados al tomar las imágenes de entrenamiento de *bunch*120 de forma aleatoria. Hemos incluido los resultados de la Prueba 1 que repite los valores del Cuadro 4.7 para facilitar la comparación. A su vez, la Figura 4.18 muestra los errores de entrenamiento y validación.

Cuadro 4.9. Exhaustividad y mAP de modelos al entrenar con imágenes originales de *bunch*120, manteniendo fijo el conjunto de validación y tomando de forma aleatoria las imágenes del conjunto de entrenamiento.

	Prueba 1	Prueba 2	Prueba 3	Prueba 4	Prueba 5
Detectados correct.	482	453	471	466	506
No detectados	159	188	170	175	135
Exhaustividad	**75,2** %	**70,67** %	**73,48** %	**72,7** %	**78,94** %
Otras detecciones	75	72	66	37	111
mAP	**0,628331**	**0,616076**	**0,660746**	**0,653352**	**0,642251**

Vemos que todas las pruebas tienen comportamientos similares, por lo que podemos considerar que la NN no depende del orden en que tiene acceso a las imágenes de entrenamiento.

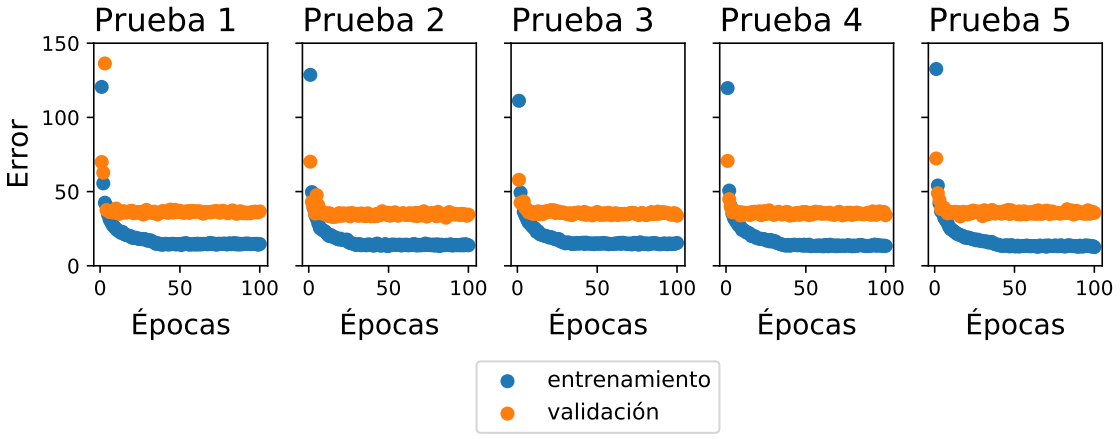


Figura 4.18. Errores de entrenamiento y validación, al tomar las imágenes de entrenamiento de *bunch*120 de forma aleatoria.

4.5.2. Conjuntos de entrenamiento y validación aleatorios

De manera similar, en el Cuadro 4.10 se observan los valores de exhaustividad y mAP de los modelos seleccionados al elegir de forma aleatoria las imágenes de entrenamiento y validación de *bunch*120. También es este caso hemos incluido los resultados de la Prueba 1 que repite los valores del Cuadro 4.7 para facilitar la comparación. Además incluimos la cantidad de racimos etiquetados en los conjuntos de entrenamiento y validación, luego de la separación aleatoria. Por otro lado, la Figura 4.19 muestra los errores de entrenamiento y validación.

Cuadro 4.10. Exhaustividad y mAP de modelos al entrenar con imágenes originales de *bunch*120, eligiendo de forma aleatoria las imágenes del conjunto de entrenamiento y validación.

	Prueba 1	Prueba 2	Prueba 3	Prueba 4	Prueba 5
Entrenamiento	593	584	593	575	582
Validación	102	111	102	120	113
Detectados correct.	482	563	538	581	489
No detectados	159	78	103	60	152
Exhaustividad	**75,2** %	**87,83** %	**83,93** %	**90,64** %	**76,29** %
Otras detecciones	75	141	100	193	37
mAP	**0,628331**	**0,781485**	**0,756223**	**0,818391**	**0,782055**

Notamos que en todos los casos los modelos tuvieron un mejor desempeño al mezclar las fotografías y tomar de forma aleatoria las imágenes para los conjuntos de entrenamiento y validación.

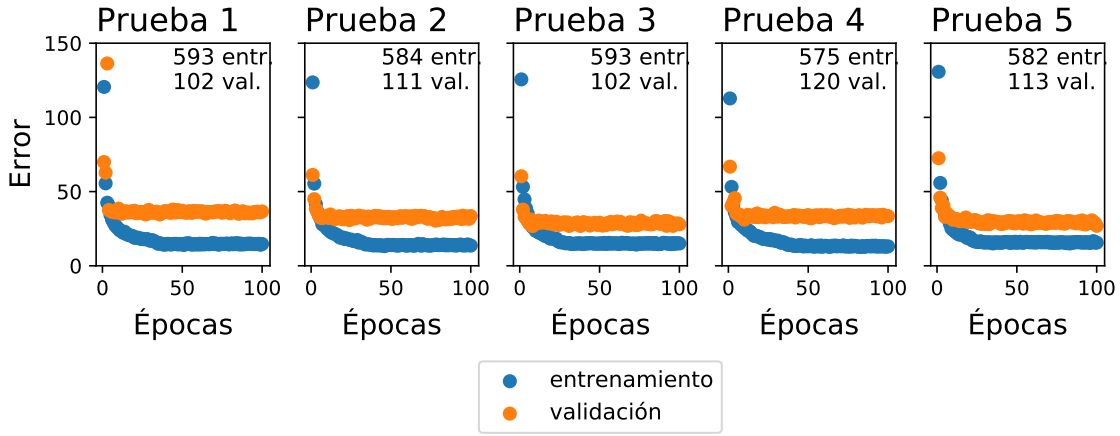


Figura 4.19. Errores de entrenamiento y validación, al seleccionar las imágenes de entrenamiento y validación de forma aleatoria.

4.6. Rendimiento computacional

Como interés secundario del trabajo medimos el rendimiento del análisis realizado en cinco infraestructuras de hardware, tanto CPUs como GPUs. Para este análisis utilizamos el conjunto de datos *bunch*120 para solamente una época, e hicimos cinco pruebas en cada infraestructura.

Como se puede ver en la Figura 4.20 y en el Cuadro 4.11 la GPU Titan Xp, como es de esperar, tiene el mejor rendimiento siendo ∼3 veces más rápida que los 64 núcleos del nodo Opteron 6376 y ∼1,4 veces mas rápida que la GPU Titan X de previa generación. Las dos GPUs tiene un rendimiento superior al CPU más rápido, el AMD Epyc 7281.

Cuadro 4.11. Tiempo medio en segundos para cinco pruebas de entrenamiento en cinco infraestructuras de CPUs y GPUs diferentes.

Hardware	Tiempo medio (s)	Desviación estándar
Opteron	1652	35
Ryzen	1080	29
Epyc	962	21
Titan X	787	7
Titan Xp	563	12

El tiempo de entrenamiento para 100 experimentos y tamaño de lote 2 en la GPU Titan Xp, usando conjuntos con 500 imágenes de entrenamiento y 100 de validación fue en promedio de ∼31 horas. En esta GPU se corrieron las pruebas originales + ruido Laplaciano, originales + pintura y originales de *bunch*600.1. A su vez, la etapa de evaluación de modelos para estas pruebas fue en promedio de ∼3,5 horas. Por otro lado, el entrenamiento de originales + reflexión (es decir, 200 imágenes de entrenamiento y 40 de validación) tomó ∼11 horas.

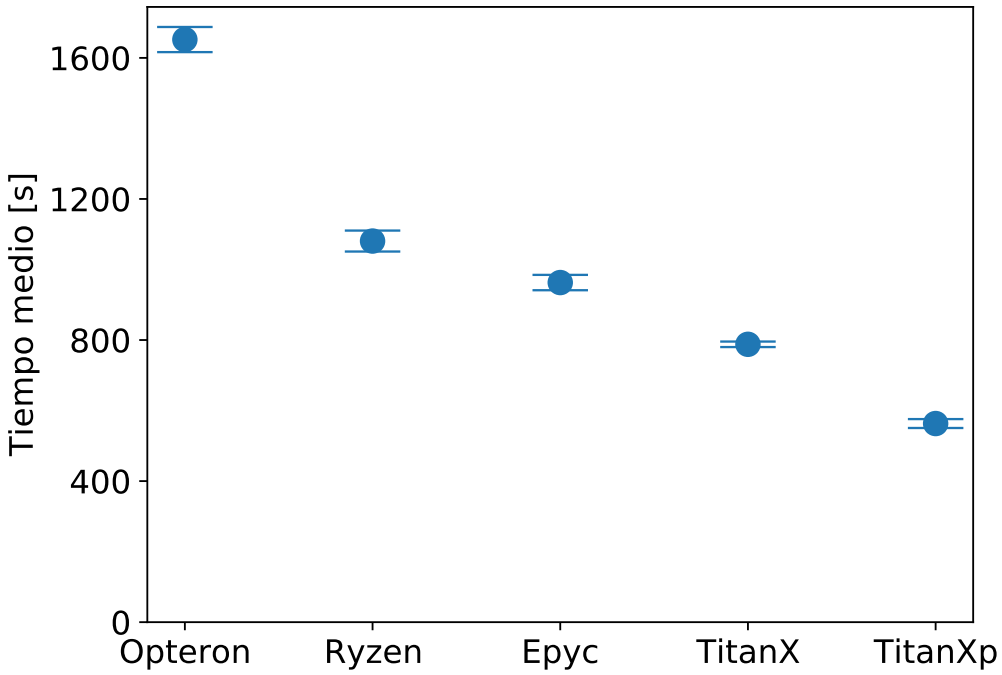


Figura 4.20. Comparación del rendimiento en diferentes infraestructuras de CPUs y GPUs.

Por su parte, el tiempo para entrenar durante 100 épocas y tamaño de lote 2 en la GPU Titan X, solo con *bunch*120 llevó ∼12 horas. El entrenamiento de originales + solarización y originales + CLAHE, que duplicaron el dataset de *bunch*120, tomaron ∼24 horas, mientras que originales + empañado, originales + ruido Gaussiano, originales + ruido de Poisson que quintuplicaron a *bunch*120, así como el entrenamiento con *bunch*600.2 llevaron ∼61 horas. Por otro lado, el entrenamiento de este último usando tamaño de lote 8 llevó ∼22 horas, mientras que entrenar con las imágenes originales de *bunch*120 usando este tamaño de lote tomó ∼5 horas.

En virtud de los tiempos de entrenamiento nos vimos limitados en la cantidad de filtros que podíamos probar, así como en la repetición de pruebas con nuevos hiperparámetos, y distintas combinaciones de filtros y transformaciones, debido en principio a que no teníamos uso exclusivo del hardware del cluster, sumado a múltiples problemas de conexión y con los equipos durante la cuarentena obligatoria.

Capítulo 5

Conclusiones

En este Capítulo recopilamos las principales conclusiones de este trabajo, así como planteamos posibles pruebas a realizar a futuro.

En este trabajo estudiamos lo que es el Aprendizaje Automático y algoritmos de aprendizaje supervisado para resolver un problema particular de detección de objetos en imágenes. Para ello vimos los elementos del aprendizaje supervisado y los tipos de tareas que se pueden resolver con éste. Luego profundizamos en Redes Neuronales, su inspiración biológica, definición formal y estructura para comprender el método de gradiente descendente y propagación hacia atrás para entrenar a una red, para luego hacer foco en Redes Neuronales Convolucionales, los tipos de capas que la caracterizan, transferencia de aprendizaje y aprendizaje residual, ya que este tipo de red es el que se suele usar en tareas que impliquen el análisis de imágenes. Entonces describimos en qué consiste la detección de objetos en imágenes, y cómo se mide el rendimiento de los algoritmos que resuelven esta tarea, en particular vimos en detalle la estructura de la red YOLOv3, así como su función error característica. Finalmente pusimos a prueba lo estudiado en imágenes de campo, capturadas en fincas del Valle de Uco y Luján de Cuyo, las cuales etiquetamos y separamos en dos conjuntos principales de 100 y 500 imágenes de entrenamiento. Aplicamos técnicas de aumento de datos al primero de estos conjuntos y comparamos lo obtenido en estas pruebas con los resultados de entrenar con el conjunto más numeroso, evaluando mAP de los modelos y calculando exhaustividad y precisión de las detecciones. Además pusimos a prueba diferentes valores de hiperparámetros y medimos el rendimiento computacional de la red en distintas infraestructuras de hardware.

5.1. Objetivos cumplidos

En este Seminario de Investigación y/o Desarrollo Tecnológico estudiamos algoritmos supervisados para la detección de objetos en imágenes. Si bien uno de los objetivos era detectar granos de uva, luego de las reuniones mantenidas con productores y técnicos del INTA y COVIAR, pusimos a prueba YOLOv3 para detectar racimos de uva en imágenes de campo.

Estudiamos diversas medidas de rendimiento de los algoritmos, para los diferentes tipos de tareas a resolver por los algoritmos de aprendizaje supervisado. En particular examinamos el valor mAP calculado por la biblioteca `ImageAI` para cada modelo, así como calculamos la exhaustividad y precisión para comparar el desempeño de la red al ser entrenada con diferentes datasets.

Estudiamos y analizamos la matemática subyacente de los algoritmos, en particular profundizamos en Redes Neuronales y Redes Neuronales Convolucionales para lograr comprender cómo opera la red YOLOv3. A partir de esto realizamos pruebas con distintos valores de los hiperparámetros tamaño de lote y tasa de aprendizaje, y encontramos una combinación de estos ($\texttt{batch\_size} = 8$ y $\texttt{LR} = 1 \times 10^{-4}$) que mejoran el desempeño de la red, teniendo en cuenta las limitaciones de memoria de las GPUs utilizadas.

Aplicamos lo desarrollado en la tesis sobre imágenes de campo capturadas en fincas del Valle de Uco y de Luján de Cuyo, logrando un 96,1 % de racimos detectados correctamente con una precisión del 72,16 % para el modelo entrenado a partir de *bunch*600.

Respecto a las hipótesis formuladas, podemos concluir que es posible utilizar algoritmos de AA para identificar de forma confiable en imágenes los frutos de la planta de la vid. Además es viable mejorar los resultados obtenidos y el tiempo de cómputo necesario para obtenerlos, mediante la optimización del espacio de hiperparámetros de los modelos. Sin embargo, no pusimos a prueba la segunda hipótesis, ya que luego de las reuniones con productores y técnicos del INTA y COVIAR en las que manifestaron que para ellos es más importante saber la cantidad de racimos que contar granos de uva, hicimos foco en la detección de racimos los cuales pudimos cuantificar con una precisión de entre el 70 y el 82 % según el modelo utilizado.

Además logramos implementar Redes Neuronales en infraestructura de Computación de Alto Desempeño (HPC), y analizar el rendimiento de los algoritmos en varias arquitecturas HPC. Como el acceso a éstos era compartido con otros usuarios no pudimos completar todas las etapas como en las demás pruebas, esto es entrenamiento con 50 o 100 experimentos, evaluación de modelos y detección, por lo que no pudimos comparar la exhaustividad y precisión en estos casos. Sin embargo, al entrenar con una sola época en cinco repeticiones pudimos comparar el tiempo medio de ejecución, el cual es extrapolable a un número mayor de experimentos.

5.2. Principales resultados

Si se cuenta con una cantidad limitada de imágenes como en el caso del entrenamiento con las fotos originales de *bunch*120, es posible aumentar la exhaustividad del modelo de detección al aplicar técnicas de aumento de datos. De toda la batería de pruebas realizadas, es notable remarcar que con sólo aplicar la transformación de reflexión horizontal se incrementa más la exhaustividad y precisión, en comparación a lo alcanzado en las pruebas que quintuplicaron el dataset original. Sin embargo ninguna de estas pruebas logró alcanzar o superar el rendimiento obtenido con *bunch*600, lo cual puede explicarse porque este último conjunto tiene mayor número de fotografías distintas entre sí, en cambio el trabajar con las técnicas de aumento de datos elegidas genera pequeñas diferencias con las imágenes originales, por lo que las capas convolucionales no podrían extraer un mayor número de características de las imágenes transformadas, especialmente del agregado de ruido. Además, en algunos casos, el aplicar estas técnicas aumentó la cantidad de detecciones con niveles de certeza mayores al 50 %.

A pesar de lo esperado, quintuplicar el conjunto de datos no siempre lleva a mejores resultados. Por un lado las aplicaciones de ruido Gaussiano y Laplaciano obtuvieron un mAP muy bajo, mientras que el ruido de Poisson y el empañado tuvieron una exhaustividad apenas por debajo de la alcanzada al entrenar solamente con las imágenes originales de *bunch*120.

Por otro lado, a la hora de entrenar es importante tener en cuenta que el algoritmo utilizado no depende del orden en que son ingresadas las imágenes del conjunto de entrenamiento, aunque sí es aconsejable hacer la separación de imágenes para dicho conjunto y el de validación de forma aleatoria. Además, es posible mejorar el tiempo de cómputo y los resultados obtenidos al modificar los valores de los hiperparámetros, especialmente el tamaño de lote. Si bien hicimos pruebas con 50 y 100 experimentos, es probable que si usáramos entre 60 y 80 épocas bastaría para que los errores se estabilicen, lo cual implicaría menos tiempo.

Respecto a las deficiencias que obtuvimos, podríamos solucionar el sobreajuste con la obtención de un mayor número de fotografías, aunque esto implicaría mucho más tiempo para entrenar la red; mientras que la detección de un mismo racimo más de una vez probablemente se deba a una mala etiquetación de los racimos, punto que nos hizo notar el Ingeniero Agrónomo Marcos Montoya del INTA, por lo que podríamos re-etiquetar las imágenes y observar si hay mejoras en las detecciones.

5.3. Trabajo futuro

Como trabajo futuro, quisiéramos entrenar nuevas Redes Neuronales con el objetivo de detectar racimos de uva en imágenes y videos capturados por drones volando sobre los espalderos, a unos 3 o 4 metros por encima del suelo. Esto lo haríamos para diferentes

estadios de la planta, especialmente para los primeros estadios de fructificación, ya que esto sería aún más provechoso para lograr un pronóstico de cosecha temprano.

Además quisiéramos aplicar las mejores transformaciones y filtros a las imágenes de *bunch*600, y repetir las pruebas de aumento de datos con los mejores valores de hiperparámetros encontrados, así como probar nuevas transformaciones, como reflexión vertical y rotaciones, ya que vimos que la reflexión horizontal logró muy buenos resultados. También quisiéramos formar una gran base de datos a partir de las imágenes originales más todas las transformaciones y filtros utilizados, para tomar distintas combinaciones aleatorias de allí y probar el comportamiento de la red con un conjunto de entrenamiento formado a partir de varias técnicas de aumento de datos a la vez.

Finalmente, queremos remarcar el impacto que ha tenido este trabajo a nivel local, ya que hemos despertado el interés de distintos productores y bodegas, lo cual es para mí una motivación importante para seguir formándome e investigando. Si bien abordar este tema representó un desafío personal por requerir el aprendizaje de muchos temas ajenos a mi formación de base, el haber alcanzado los resultados obtenidos, en gran parte en un contexto tan particular como lo es la cuarentena, y que éstos hayan tenido un impacto en el medio, me anima a continuar y puedo decir con alegría que mereció el esfuerzo y que quiero seguir profundizando en Aprendizaje Automático, un campo de estudio del cual no tenía conocimiento hasta que comencé con este Seminario de Investigación y/o Desarrollo Tecnológico.

Apéndice A

Aplicación de técnicas de aumento de datos

A.1. Ejemplo script de Bash

En el Código A.1 vemos un ejemplo de aplicación de técnicas de aumento de datos. Con un script de `Bash` creo los directorios correspondientes, uso `ImageMagick` para aplicar la transformación (en este caso particular, CLAHE) a cada imagen y modifico la línea `<filename>` de cada archivo `.xml`.

```
#!/bin/bash

cd /home/tparlanti/JAIIO/bunch120/training/originales

CARPETAS=("train" "validation")
IMAGENES_DISTORSIONADAS=../distorsiones

# Crea directorios
mkdir -p $IMAGENES_DISTORSIONADAS/Clahe4/train/images
    $IMAGENES_DISTORSIONADAS/Clahe4/train/annotations
    $IMAGENES_DISTORSIONADAS/Clahe4/validation/images
    $IMAGENES_DISTORSIONADAS/Clahe4/validation/annotations > /dev/null 2> /
    dev/null

for carpeta in "${CARPETAS[@]}"; do
    for img in `ls ./$carpeta/images`; do
        # Usar ImageMagick para aplicar clahe a las imágenes
        magick ./$carpeta/images/$img -clahe 5x5%+128+4
    $IMAGENES_DISTORSIONADAS/Clahe4/${carpeta}/images/${img%.*}_clahe4.jpg
        # Copiar los archivos .xml con las etiquetas
        cp ./$carpeta/annotations/${img%.*}.xml $IMAGENES_DISTORSIONADAS/
    Clahe4/$carpeta/annotations/${img%.*}_clahe4.xml
        # Modificar la línea <filename>
```

```
        sed -i "s%.*filename.*%        <filename>${img%.*}_clahe4.jpg</filename>%" $IMAGENES_DISTORSIONADAS/Clahe4/$carpeta/annotations/${img%.*}_clahe4.xml
    done
done
```

Código A.1. Script de bash para transformar imágenes y los correspondientes archivos .xml.

A.2. Scripts de Bash y Python para espejar

Para el caso particular del espejado de imágenes, además de un script de `Bash` similar al anterior, uso un script de `Python` para modificar las coordenadas de los vértices de los rectángulos etiquetados. En este caso, dado que estoy editando los archivos `.xml` desde `Python`, modifico la línea `<filename>` en esa misma edición. Vemos los scripts en los Códigos A.2 y A.3, respectivamente.

```
#!/bin/bash


cd /home/tparlanti/JAIIO/bunch120/training/originales

CARPETAS=("train" "validation")
IMAGENES_DISTORSIONADAS=../distorsiones #para bunch120

# Crea directorios
mkdir -p $IMAGENES_DISTORSIONADAS/Espejo/train/images $IMAGENES_DISTORSIONADAS/Espejo/train/annotations $IMAGENES_DISTORSIONADAS/Espejo/validation/images $IMAGENES_DISTORSIONADAS/Espejo/validation/annotations > /dev/null 2> /dev/null

for carpeta in "${CARPETAS[@]}"; do
    PATH_ANNOTATIONS_ORIGINALES=./$carpeta/annotations
    PATH_ANNOTATIONS_ESPEJO=$IMAGENES_DISTORSIONADAS/Espejo/$carpeta/annotations
    # Ejecutar script de Python, pasando los valores anteriores, para modificar las coordenadas de los vértices en los archivos .xml
    python3 /home/tparlanti/JAIIO/scripts/Espejo.py $PATH_ANNOTATIONS_ORIGINALES $PATH_ANNOTATIONS_ESPEJO
    # Usar ImageMagick para espejar cada imagen
    for img in `ls ./$carpeta/images`; do
        convert ./$carpeta/images/$img -flop $IMAGENES_DISTORSIONADAS/Espejo/${carpeta}/images/${img%.*}_espejo.jpg
    done
done
```

Código A.2. Script de bash para espejar imágenes y ejecutar script de Python.

```
import cv2
import sys
import os

# Valores definidos en el script de bash
PATH_ANNOTATIONS_ORIGINALES = sys.argv[1]
PATH_ANNOTATIONS_ESPEJO = sys.argv[2]


def etiquetasEspejo(dirAnnotations,dirAnnotationsEspejo):
    """
    Función que espeja las coordenadas de los vértices de los rectángulos etiquetados
    y cambia la línea <filename>
    Parámetros:
        dirAnnotations: directorio con archivos .xml originales
        dirAnnotationsEspejo: directorio donde guardará los archivos .xml con coordenadas espejadas
    """
    xml = os.listdir(dirAnnotations)
    interesX=["xmin","xmax"]
    for i in range(len(xml)):
        xminOrig=[]
        xmaxOrig=[]
        leer=open(os.path.join(dirAnnotations,str(xml[i])), 'r')
        for line in leer:
            if "<width>" in line:
                list=line.split()
                for k in list:
                    interes="</width>"
                    lin=k.strip(interes)
                    ancho=int(lin)
            if line.strip().startswith("<xmin>"):
                l=line.strip()
                interes2="</xmin>"
                xminOrig.append(int(l.strip(interes2)))
            elif line.strip().startswith("<xmax>"):
                l=line.strip()
                interes2="</xmina>"
                xmaxOrig.append(int(l.strip(interes2)))
        leer.close()
        ListXminOrig=[]
        ListXmaxOrig=[]
        ListXminEspejo=[]
        ListXmaxEspejo=[]
        with open(os.path.join(dirAnnotations,str(xml[i])), 'r') as leer, open(os.path.join(dirAnnotationsEspejo,str(xml[i].replace(".xml","_espejo.xml"))),"w") as escribir:
            for line in leer:
```

```
                for t in range(len(xminOrig)):
                    strmin="<xmin>"+str(xminOrig[t])+"</xmin>"
                    strmax="<xmax>"+str(xmaxOrig[t])+"</xmax>"
                    if strmin in line:
                        ListXminOrig.append(strmin)
                        xminEspejo=ancho-xmaxOrig[t]
                        ListXminEspejo.append("            <xmin>"+str(xminEspejo)+"</xmin>"+"\n")
                    if strmax in line:
                        ListXmaxOrig.append(strmax)
                        xmaxEspejo=ancho-xminOrig[t]
                        ListXmaxEspejo.append("            <xmax>"+str(xmaxEspejo)+"</xmax>"+"\n")
                if "<filename>" in line:
                    escribir.write("    <filename>"+str(xml[i].replace(".xml","_espejo.jpg"))+"</filename>"+"\n")
                elif any(word in line for word in ListXminOrig):
                    escribir.write(ListXminEspejo[ListXminOrig.index(line.split()[0])])
                elif any(word in line for word in ListXmaxOrig):
                    escribir.write(ListXmaxEspejo[ListXmaxOrig.index(line.split()[0])])
                else:
                    escribir.write(line)
        escribir.close()
        leer.close()



etiquetasEspejo(dirAnnotations=str(PATH_ANNOTATIONS_ORIGINALES), dirAnnotationsEspejo=str(PATH_ANNOTATIONS_ESPEJO))
```

Código A.3. Script de Python para editar archivos .xml

Apéndice B

Entrenamiento, evaluación y detección

B.1. Ejemplo scripts de Bash y Python

Para las etapas de entrenamiento, validación y detección, usé un script de `Bash` "base", que vemos en el Código B.1, el cual edito para cada prueba al determinar el análisis que voy a desarrollar (por ejemplo, si es con *bunch*120 o con *bunch*600, si se trata de imágenes originales u originales más transformadas), así como los valores de hiperparámetros a fijar. Con este script además ejecuto el script de `Python` que vemos en el Código B.2, al cual le paso todos los valores antes mencionados. Así, para cada prueba uso siempre el mismo script de `Python`, y solo edito el script de `Bash`.

```
#!/bin/bash

DIRECTORIO=`pwd`

SET_IMAGENES="bunch120" #bunch600

PATH_IMAGENES="${DIRECTORIO}/${SET_IMAGENES}/training/"
PATH_DETECTAR="${DIRECTORIO}/${SET_IMAGENES}/"
PATH_ANS="${DIRECTORIO}/answers"
TIPO_ANALISIS="originales" #originales, origAndEspejo, origAndClahe ...
TIPO_DETECCION="detect-cerca-2"

# Hiperparámetros:
PATH_PREENTRENADO="${DIRECTORIO}/pretrained-yolov3.h5"
ETIQUETA="bunch"
BATCH=8
IOU=0.5
OBJETO=0.3
NMS=0.5
EXPERIMENTOS=100
```

```
21
22 ANCHO=`tput cols`
23
24
25 DESTINO=${DIRECTORIO}/answers/${SET_IMAGENES}/${TIPO_ANALISIS}/${
       TIPO_ANALISIS}-${EXPERIMENTOS}_experimentos-${BATCH}batchSize
26 mkdir -p $DESTINO > /dev/null 2> /dev/null
27
28 cd $DESTINO
29
30 mkdir ./scripts
31
32 # Ejecutar script de Python, pasando los valores anteriores
33 python3 $DIRECTORIO/scripts/completo.py $ANCHO $PATH_IMAGENES
       $PATH_DETECTAR $DESTINO $PATH_PREENTRENADO $ETIQUETA $BATCH $IOU
       $OBJETO $NMS $TIPO_ANALISIS $EXPERIMENTOS $TIPO_DETECCION |& tee ans-${
       SET_IMAGENES}-${EXPERIMENTOS}exp-${TIPO_ANALISIS}-${TIPO_DETECCION}.txt
34
35 cp $DIRECTORIO/scripts/completo.sh ./scripts/completo.sh
36 cp $DIRECTORIO/scripts/completo.py ./scripts/completo.py
```

Código B.1. Script de Bash editable para etapas de entrenamiento, evaluación y detección.

```
1  from imageai.Detection.Custom import DetectionModelTrainer
2  from imageai.Detection.Custom import CustomObjectDetection
3  import os
4  import numpy as np
5  import time
6  import sys # Para tomar los definidos en el script de bash
7
8
9  os.environ["CUDA_VISIBLE_DEVICES"]="1"
10
11 ANCHO = sys.argv[1]
12
13 print("=" * int(ANCHO))
14
15 print(sys.argv[0])
16
17 PATH_IMAGENES = sys.argv[2]
18 print("PATH_IMAGENES = ", PATH_IMAGENES)
19
20 PATH_DETECTAR = sys.argv[3]
21 print("PATH_DETECTAR = ", PATH_DETECTAR)
22
23 DESTINO = sys.argv[4]
24 print("DESTINO = ", DESTINO)
25
```

```
26 PATH_PREENTRENADO = sys.argv[5]
27 print("PATH_PREENTRENADO = ", PATH_PREENTRENADO)
28
29 ETIQUETA = sys.argv[6]
30 print("ETIQUETA = ", ETIQUETA)
31
32 BATCH = sys.argv[7]
33 print("BATCH = ", BATCH)
34
35 IOU = sys.argv[8]
36 print("IOU = ", IOU)
37
38 OBJETO = sys.argv[9]
39 print("OBJETO = ", OBJETO)
40
41 NMS = sys.argv[10]
42 print("NMS = ", NMS)
43
44 TIPO_ANALISIS = sys.argv[11]
45 print("TIPO_ANALISIS = ", TIPO_ANALISIS)
46
47 EXPERIMENTOS = sys.argv[12]
48 print("EXPERIMENTOS = ", EXPERIMENTOS)
49
50 TIPO_DETECCION = sys.argv[13]
51 print("TIPO_DETECCION = ", TIPO_DETECCION)
52
53 print("=" * int(ANCHO))
54
55
56 start = time.time()
57 startPT = time.process_time()
58
59
60 def entrenamiento(dir_imagenes,tipo_analisis,dir_destino,experimentos,
       etiqueta,path_preentrenado,batch=2):
61     """
62     Entrena la red. Parámetros:
63     dir_imagenes: directorio con las imágenes
64     tipo_analisis: prueba a realizar (originales, originales con distorsión
       )
65     dir_destino: directorio donde ImageAI creará carpetas y guardará los
       modelos generados
66     experimentos: número de experimentos o épocas
67     etiqueta: clases con las que se va a entrenar
68     path_preentrenado: ruta al modelo preentrenado
69     batch: tamaño de lote, valor por defecto 2
70     """
71     dirImagenes = os.path.join(str(dir_imagenes),str(tipo_analisis))
```

```
72      for item in os.listdir(str(dirImagenes)):
73          os.symlink(os.path.join(str(dirImagenes),item), os.path.join(str(
    dir_destino),item))
74      trainer = DetectionModelTrainer()
75      trainer.setModelTypeAsYOLOv3()
76      trainer.setDataDirectory(data_directory=str(dir_destino))
77      trainer.setTrainConfig(object_names_array=[str(etiqueta)], batch_size=
    batch, num_experiments=int(experimentos), train_from_pretrained_model=
    str(path_preentrenado))
78      trainer.trainModel()
79
80
81
82
83  def eleccion_modelo(dir_destino,iou=0.5, objeto=0.3, nms=0.5):
84      """
85      Evalúa los modelos y selecciona el de mayor mAP. Parámetros:
86      dir_destino: directorio donde ImageAI creó carpetas y guardó los
    modelos generados
87      iou: umbral de IoU, valor por defecto 0.5
88      objeto: umbral OBJ de nivel de certeza, valor por defecto 0.3
89      nms: umbral de NMS, valor por defecto 0.5
90      """
91      trainer = DetectionModelTrainer()
92      trainer.setModelTypeAsYOLOv3()
93      trainer.setDataDirectory(data_directory=str(dir_destino))
94      metrics = trainer.evaluateModel(model_path=os.path.join(str(dir_destino
    ), "models"), json_path=os.path.join(str(dir_destino),"json/
    detection_config.json"), iou_threshold=iou, object_threshold=objeto,
    nms_threshold=nms)
95      mayor_map = max(metrics, key = lambda x:x['map'])
96      M = mayor_map['map']
97      bool = []
98      for j in range(len(metrics)):
99          bool.append(metrics[j]['map'] == M)
100     if bool.count(True) == 1:
101         modelo_elegido = mayor_map['model_file']
102         datos = mayor_map
103     else:
104         ies = np.where(bool)[0]
105         lista = [float(metrics[k]['model_file'].replace(".h5", "").split("-
    ")[-1]) for k in ies]
106         loss = [{'loss': valor} for valor in lista]
107         metrics2 = metrics.copy()
108         metrics2 = [metrics2[k] for k in ies]
109         assert(len(metrics2) == len(loss))
110         for l in range(len(metrics2)):
111             metrics2[l].update(loss[l])
112         menor_loss = min(metrics2, key = lambda x:x['loss'])
```

```
113         modelo_elegido = menor_loss['model_file']
114         datos = menor_loss
115     print(metrics)
116     return (modelo_elegido, datos)
117
118
119
120
121 def deteccion(dir_destino,modelo_elegido,dir_imagenes_detectar,tipo_det,
    tipo_analisis):
122     """
123     Pone a prueba el modelo seleccionado en la función anterior con las imá
    genes de detect. Parámetros:
124     dir_destino: directorio donde ImageAI creó carpetas y guardó los
    modelos generados
125     modelo_elegido: modelo seleccionado de la etapa de evaluación
126     dir_imagenes_detectar, tipo_det: directorio con imágenes de detect
127     tipo_analisis: prueba que estamos comparando (originales, originales
    con distorsión)
128     """
129     detector = CustomObjectDetection()
130     detector.setModelTypeAsYOLOv3()
131     detector.setModelPath(str(modelo_elegido)) #completa con mejor modelo
132     detector.setJsonPath(os.path.join(str(dir_destino),"json/
    detection_config.json"))
133     detector.loadModel()
134     dirDetectar = os.path.join(dir_imagenes_detectar,tipo_det,"images")
135     detectados = str(dir_destino) + "/detected-" + str(tipo_det)
136     os.makedirs(detectados, exist_ok = True)
137     for file in os.listdir(str(dirDetectar)):
138         f = os.path.join(str(dirDetectar),file)
139         f1 = str(detectados) + "/detected-" + str(tipo_analisis) + "-" +
    file
140         detections = detector.detectObjectsFromImage(input_image = str(f),
    output_image_path = str(f1), minimum_percentage_probability=30)
141         print(" usando ", modelo_elegido)
142         if detections:
143             for detection in detections:
144                 print(file, " : ", detection["name"], " : ", detection["
    percentage_probability"], " : ", detection["box_points"])
145         else:
146             print(file, " : ", "sin detectar")
147
148
149
150
151 #
    ##############################################################################
```

```
152
153
154
155 #Primera funcion:
156 startPT1=startPT
157 start1=start
158
159 entrenamiento(dir_imagenes=str(PATH_IMAGENES),tipo_analisis=str(
    TIPO_ANALISIS),dir_destino=str(DESTINO),experimentos=str(EXPERIMENTOS),
    etiqueta=str(ETIQUETA),path_preentrenado=str(PATH_PREENTRENADO),batch=
    int(BATCH))
160
161
162 end1=time.time()
163 endPT1=time.process_time()
164
165 time1=end1-start1
166 timePT1=endPT1-startPT1
167
168 print("time1: ", time1, " | process_time1: ", timePT1)
169 print("=" * int(ANCHO))
170
171
172 #Segunda funcion:
173 start2=time.time()
174 startPT2=time.process_time()
175
176
177 (que_modelo, datos_modelo)=eleccion_modelo(dir_destino=str(DESTINO),iou=
    float(IOU), objeto=float(OBJETO), nms=float(NMS))
178 print("\n", datos_modelo, "\n")
179 print(que_modelo)
180
181 end2=time.time()
182 endPT2=time.process_time()
183
184 time2=end2-start2
185 timePT2=endPT2-startPT2
186
187 print("time2: ", time2, " | process_time2: ", timePT2)
188 print("=" * int(ANCHO))
189
190 #Tercera funcion:
191 start3=time.time()
192 startPT3=time.process_time()
193
194 deteccion(dir_destino=str(DESTINO),modelo_elegido=str(que_modelo),
    dir_imagenes_detectar=str(PATH_DETECTAR),tipo_det=str(TIPO_DETECCION),
    tipo_analisis=str(TIPO_ANALISIS))
```

```
195
196 end3=time.time()
197 endPT3=time.process_time()
198
199 time3=end3-start3
200 timePT3=endPT3-startPT3
201
202 print("time3: ", time3, " | process_time3: ", timePT3)
203 print("=" * int(ANCHO))
204
205 end=time.time()
206 endPT=time.process_time()
207 timetotal=end-start
208 timetotalPT=endPT-startPT
209 print("Total_time: ", timetotal, "| Total_process_time: ", timetotalPT)
```

Código B.2. Script de Python para entrenar, evaluar y detectar.

Apéndice C

Análisis de resultados

C.1. Script de Python

Al igual que para llevar a cabo las etapas de entrenamiento, evaluación y detección, uso un script de `Bash` que edito según la prueba a analizar que ejecuta siempre el mismo script de `Python`, el cual vemos en el Código C.1.

```
import os
import json
import cv2
import pprint
import numpy as np
import sys
import matplotlib as mpl
import matplotlib.pyplot as plt



ANCHO = sys.argv[1]
print("\n","=" * int(ANCHO),"\n")


print(sys.argv[0])

SET_IMAGENES = sys.argv[2]
print("SET_IMAGENES = ", SET_IMAGENES)

TIPO_ANALISIS = sys.argv[3]
print("TIPO_ANALISIS = ", TIPO_ANALISIS)

TIPO_DETECCION = sys.argv[4]
print("TIPO_DETECCION = ", TIPO_DETECCION)

EXPERIMENTOS = sys.argv[5]
print("EXPERIMENTOS = ", EXPERIMENTOS)
```

```
29
30 BATCH = sys.argv[6]
31 print("BATCH = ", BATCH)
32
33 PATH_DETECTAR = sys.argv[7]
34 print("PATH_DETECTAR = ", PATH_DETECTAR)
35
36 PORCENTAJE_INTERSECCION = sys.argv[8]
37 print("PORCENTAJE_INTERSECCION = ", PORCENTAJE_INTERSECCION)
38
39 SUMA_AREAS = sys.argv[9]
40 print("SUMA_AREAS = ", SUMA_AREAS)
41
42 DESTINO = sys.argv[10]
43 print("DESTINO = ", DESTINO)
44
45 NOMBRE_ARCHIVO = sys.argv[11]
46 print("NOMBRE_ARCHIVO = ", NOMBRE_ARCHIVO)
47
48 NUM_IMG_TRAIN = sys.argv[12]
49 print("NUM_IMG_TRAIN = ", NUM_IMG_TRAIN)
50
51 NUM_IMG_VALIDATION = sys.argv[13]
52 print("NUM_IMG_VALIDATION = ", NUM_IMG_VALIDATION)
53
54 MODELO = sys.argv[14]
55 print("MODELO = ", MODELO)
56
57 MAP = sys.argv[15]
58 print("MAP = ", MAP)
59
60 LR = sys.argv[-1]
61 if LR == MAP:
62     LR = None
63 else:
64     print("LR = ", LR)
65
66 print("\n","=" * int(ANCHO),"\n")
67
68
69
70 def etiquetas(dir_detect,tipo_det):
71     """
72     Lee los archivos .xml de cada imagen y almacena las coordenadas de los
    rectángulos etiquetados en un diccionario.
73     Parámetros: directorio con las imágenes de detect
74     """
75     xmldetectar = os.listdir(os.path.join(dir_detect,tipo_det,"annotations"
    ))
```

```
76      etiquetasDict={}
77      interes=["xmin","ymin","xmax","ymax"]
78      for i in range(len(xmldetectar)):
79          etiquetasDict[xmldetectar[i].replace("xml","jpg")]=[]
80          leer=open(os.path.join(dir_detect,tipo_det,"annotations",str(
    xmldetectar[i])),"r")
81          for j in leer:
82              if "<bndbox>" in j:
83                  lista=[]
84              for t in range(len(interes)):
85                  if interes[t] in j:
86                      list=j.split()
87                      for s in list:
88                          interes2="</xymina>"
89                          lin=s.strip(interes2)
90                          lista.append(int(lin))
91                      if len(lista)==4:
92                          etiquetasDict[xmldetectar[i].replace("xml","jpg")].
    append(lista)
93          leer.close()
94      return etiquetasDict
95
96
97
98  def detectados(dir_detect,tipo_det,archivo_ans):
99      """
100     Lee el archivo .txt con la salida impresa luego de la detección y
    almacena las coordenadas de los rectángulos detectados y el nivel de
    certeza en un diccionario
101     Parámetros: directorio con las imágenes de detect y archivo con la
    salida de las detecciones
102     """
103     imgdetectar=os.listdir(os.path.join(dir_detect,tipo_det,"images"))
104     detectadosDict={}
105     leer2=open(archivo_ans,"r")
106     lines=leer2.readlines()
107     for i in range(len(imgdetectar)):
108         detectadosDict[imgdetectar[i]]=[]
109         for j in reversed(lines):
110             if imgdetectar[i] in j:
111                 linea=j.split(":")
112                 linea1=linea[-1].strip()
113                 if linea1=="sin detectar":
114                     linea11=[]
115                     linea2=0
116                     detectadosDict[imgdetectar[i]].append([linea11,linea2])
117                 else:
118                     linea2=linea[-2].strip()
119                     try:
```

```
120                         detectadosDict[imgdetectar[i]].append([json.loads(
    linea1),json.loads(linea2)])
121                     except:
122                         print("Error en ", archivo_ans, " : ", imgdetectar[
    i])
123                         arreglar=input("Ingrese SI o NO para arreglar el
    error: ")
124                         if arreglar.upper() == "SI":
125                             coordenadas = input("Ingrese lista con
    coordenadas detectadas: ")
126                             certeza = input("Ingrese nivel de certeza de
    deteccion: ")
127                             try:
128                                 detectadosDict[imgdetectar[i]].append([json
    .loads(coordenadas), json.loads(certeza)])
129                             except:
130                                 print("No se pudo arreglar")
131                         else:
132                             continue
133     leer2.close()
134     return detectadosDict
135
136
137
138 def calculateIntersection(a0,a1,b0,b1):
139     """
140     Calcula intersección entre los intervalos A = [a0, a1] y B = [b0, b1]
141     """
142     if a0>=b0 and a1<=b1:
143         intersection=a1-a0
144     elif a0<b0 and a1>b1:
145         intersection=b1-b0
146     elif a0<b0 and a1>b0:
147         intersection=a1-b0
148     elif a1>b1 and a0<b1:
149         intersection=b1-a0
150     else:
151         intersection=0
152     return intersection
153
154
155
156 def comparaciones(dir_detect,tipo_det,etiquetasDict,detectadosDict,
    porcentaje_interseccion_minima):
157     """
158     Calcula intersección entre rectángulos etiquetados y detectados y
    almacena las posibles detecciones para cada rectángulo etiquetado, así
    como las "otras detecciones" para cada imagen, en un diccionario
```

```
    Parámetros: directorio con las imágenes de detect, diccionario con las coordenadas de las etiquetas, diccionario con las coordenadas y nivel de certeza de las detecciones y porcentaje de intersección mínima de áreas (=25)
    """
    imgdetectar=os.listdir(os.path.join(dir_detect,tipo_det,"images"))
    comparaciones={}
    for i in range(len(imgdetectar)):
        comparaciones[imgdetectar[i]]={}
        for j in range(len(etiquetasDict[imgdetectar[i]])):
            comparaciones[imgdetectar[i]]['{0:03d}'.format(j+1) + ") rec_" + str(j+1)]={}
            comparaciones[imgdetectar[i]]['{0:03d}'.format(j+1) + ") rec_" + str(j+1)]["1) rec_esperado"]=etiquetasDict[imgdetectar[i]][j]
            X0,Y0,X1,Y1=etiquetasDict[imgdetectar[i]][j]
            AREA=float((X1-X0)*(Y1-Y0))
            rectangles=[]
            for k in range(len(detectadosDict[imgdetectar[i]])):
                comparaciones[imgdetectar[i]]['{0:03d}'.format(j+1) + ") rec_" + str(j+1)]["2) rec_detectados"]=[]
                comparaciones[imgdetectar[i]]['{0:03d}'.format(j+1) + ") rec_" + str(j+1)]["3) porcentajeInterseccion"]=[]
                comparaciones[imgdetectar[i]]['{0:03d}'.format(j+1) + ") rec_" + str(j+1)]["4) porcentajeCerteza"]=[]
                rectangles.append(detectadosDict[imgdetectar[i]][k][0])
                try:
                    for x0,y0,x1,y1 in rectangles:
                        width=calculateIntersection(x0,x1,X0,X1)
                        height=calculateIntersection(y0,y1,Y0,Y1)
                        area=width*height
                        rate=area/AREA
                        if(rate > float(porcentaje_interseccion_minima)/100):
                        # Si la intersección de las áreas es superior al porcentaje_interseccion_minima (25%), entonces se considera detección posible asociada al rectángulo etiquetado
                            comparaciones[imgdetectar[i]]['{0:03d}'.format(j+1) + ") rec_" + str(j+1)]["2) rec_detectados"].append([x0,y0,x1,y1])
                            comparaciones[imgdetectar[i]]['{0:03d}'.format(j+1) + ") rec_" + str(j+1)]["3) porcentajeInterseccion"].append(rate*100)
                            comparaciones[imgdetectar[i]]['{0:03d}'.format(j+1) + ") rec_" + str(j+1)]["4) porcentajeCerteza"].append(next(c for c in detectadosDict[imgdetectar[i]] if c[0] == [x0,y0,x1,y1])[1])
                except:
                    continue
        # Si una detección no está asociado a un rectángulo etiquetado, se considera "otra detección"
        otras=[]
```

```
        for a in range(len(detectadosDict[imgdetectar[i]])):
            SiNo=[]
            for b in range(len(etiquetasDict[imgdetectar[i]])):
                if(detectadosDict[imgdetectar[i]][a][0] == [] or (
    detectadosDict[imgdetectar[i]][a][0] in comparaciones[imgdetectar[i]]['
    {0:03d}'.format(b+1) + ") rec_" + str(b+1)]["2) rec_detectados"])):
                    SiNo.append("SI")
                else:
                    SiNo.append("NO")
            if not ("SI" in SiNo):
                otras.append(detectadosDict[imgdetectar[i]][a])
        comparaciones[imgdetectar[i]]['{0:03d}'.format(len(etiquetasDict[
    imgdetectar[i]])+1) + ") otras_detecciones"]=otras
    return comparaciones



def otrasDet(dir_detect,tipo_det,etiquetasDict,comparaciones):
    """
    Calcula el total de "otras detecciones" y reporta diccionario de "otras
     detecciones".
    Parámetros: directorio con las imágenes de detect, diccionario con las
    coordenadas de las etiquetas, diccionario con las comparaciones entre
    rectángulos.
    """
    imgdetectar=os.listdir(os.path.join(dir_detect,tipo_det,"images"))
    sumaOtrasDetecciones=0
    otrasDetecciones={}
    for i in range(len(imgdetectar)):
        if not(comparaciones[imgdetectar[i]]['{0:03d}'.format(len(
    etiquetasDict[imgdetectar[i]])+1) + ") otras_detecciones"]==[]):
            sumaOtrasDetecciones += len(comparaciones[imgdetectar[i]]['
    {0:03d}'.format(len(etiquetasDict[imgdetectar[i]])+1) + ")
    otras_detecciones"])
            otrasDetecciones[imgdetectar[i]]=comparaciones[imgdetectar[i]][
    '{0:03d}'.format(len(etiquetasDict[imgdetectar[i]])+1) + ")
    otras_detecciones"]
    return (sumaOtrasDetecciones, otrasDetecciones)



def sumas(dir_detect,tipo_det,etiquetasDict,comparaciones,suma_areas_min,
    bien_det):
    """
    Calcula el total de racimos etiquetados y el total de racimos no
    detectados.
    Reporta los diccionarios: racimos etiquetados que no fueron detectados,
     detecciones racimos no satifactorias, detecciones correctas de racimos
```

```
    Parámetros: directorio con las imágenes de detect, diccionario con las coordenadas de las etiquetas, diccionario con las comparaciones entre rectángulos, suma mínima de intersección entre áreas (=60), y diccionario con detecciones que se corresponden con racimos
    """
    imgdetectar=os.listdir(os.path.join(dir_detect,tipo_det,"images"))
    sumaEtiquetas=0
    sumaNoDetectados=0
    noDetectados=[]
    noDetectadosD={}
    bienDetectadosD={}
    deteccionesIncorrectas={}
    deteccionesCorrectas={}
    for i in range(len(imgdetectar)):
        deteccionesIncorrectas[imgdetectar[i]]=[]
        deteccionesCorrectas[imgdetectar[i]]=[]
        bienDetectadosD[imgdetectar[i]]=[]
        sumaEtiquetas += len(etiquetasDict[imgdetectar[i]])
        for b in range(len(etiquetasDict[imgdetectar[i]])):
            if(comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]["2) rec_detectados"]==[] or sum(comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]["3) porcentajeInterseccion"]) < float(suma_areas_min)):
            # Si un racimo etiquetado no tiene posibles detectados asociados o si la suma de las intersecciones de las áreas es menor a suma_areas_min (=60), entonces se considera "no detectado"
                sumaNoDetectados += 1
                noDetectados.append(imgdetectar[i])
                noDetectadosD[imgdetectar[i]]=(noDetectados.count(imgdetectar[i]), "de", len(etiquetasDict[imgdetectar[i]]))
            else:
            # Sino, se considera "detectado correctamente"
                if(len(comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]['2) rec_detectados']) > 1):
                    correcto = []
                    for elem in range(len(comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]['2) rec_detectados'])):
                        correcto.append([comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]['2) rec_detectados'][elem], comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]['4) porcentajeCerteza'][elem]])
                else:
                    correcto = [*comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]['2) rec_detectados'], *comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]['4) porcentajeCerteza']]
                if not(correcto in bienDetectadosD[imgdetectar[i]]):
                    bienDetectadosD[imgdetectar[i]].append(correcto)
```

```
256             if not(comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]["2) rec_detectados"]==[]):
257                 if(sum(comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]["3) porcentajeInterseccion"]) < float(suma_areas_min)):
258                 # Si existe al menos una detección posible para un racimo etiquetado, pero la intersección de las áreas es menor a suma_areas_min (=60), entonces es una detección de racimo no satisfactoria
259                     if(len(comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]['2) rec_detectados']) > 1):
260                         incorrecto = []
261                         for elem in range(len(comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]['2) rec_detectados'])):
262                             incorrecto.append([comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]['2) rec_detectados'][elem], comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]['4) porcentajeCerteza'][elem]])
263                             if(not(incorrecto[elem] in bienDetectadosD[imgdetectar[i]]) and not(incorrecto[elem] in deteccionesIncorrectas[imgdetectar[i]])):
264                                 deteccionesIncorrectas[imgdetectar[i]].append(incorrecto[elem])
265                     else:
266                         incorrecto = [*comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]['2) rec_detectados'], *comparaciones[imgdetectar[i]]['{0:03d}'.format(b+1) + ") rec_" + str(b+1)]['4) porcentajeCerteza']]
267                         if(not(incorrecto in bienDetectadosD[imgdetectar[i]]) and not(incorrecto in deteccionesIncorrectas[imgdetectar[i]])):
268                             deteccionesIncorrectas[imgdetectar[i]].append(incorrecto)
269         # Si hubiese alguna detección que para un racimo es no satisfactoria, pero es correcta para otro racimo, la almacenamos solamente como correcta
270         Lrep=[]
271         for eL in range(len(deteccionesIncorrectas[imgdetectar[i]])):
272             if(deteccionesIncorrectas[imgdetectar[i]][eL] in bienDetectadosD[imgdetectar[i]]):
273                 Lrep.append(deteccionesIncorrectas[imgdetectar[i]][eL])
274         for l in range(len(Lrep)):
275             deteccionesIncorrectas[imgdetectar[i]].remove(Lrep[l])
276         for j in range(len(bien_det[imgdetectar[i]])):
277             if not bien_det[imgdetectar[i]][j] in deteccionesIncorrectas[imgdetectar[i]]:
278                 deteccionesCorrectas[imgdetectar[i]].append(bien_det[imgdetectar[i]][j])
279     return (sumaEtiquetas, sumaNoDetectados, noDetectadosD, deteccionesIncorrectas, deteccionesCorrectas)
280
```

```
281
282
283 def promedios(dir_detect,tipo_det,detectadosDict,comparaciones):
284     """
285     Función que reporta:
286         Diccionario con valor promedio y desviación estandar de los niveles
     de certezas de los racimos detectados que se corresponden con un
    racimo etiquetado, por imagen
287         Promedio del TOTAL de los niveles de certeza de las detecciones que
     se corresponden con un racimo etiquetado
288         Desviacion estandar del TOTAL de los niveles de certeza de las
    detecciones que se corresponden con un racimo etiquetado
289     Parámetros: directorio con las imágenes de detect, diccionario con las
    coordenadas de las detecciones, diccionario con las comparaciones entre
     rectángulos
290     """
291     imgdetectar=os.listdir(os.path.join(dir_detect,tipo_det,"images"))
292     certezas={}
293     lista1=[]
294     for i in range(len(imgdetectar)):
295         lista=[]
296         for c in range(len(detectadosDict[imgdetectar[i]])):
297             if not (detectadosDict[imgdetectar[i]][c] in comparaciones[
    imgdetectar[i]]['{0:03d}'.format(len(comparaciones[imgdetectar[i]])) +"
    ) otras_detecciones"]):
298                 lista.append(detectadosDict[imgdetectar[i]][c][1])
299         if lista==[]:
300                 lista.append(0)
301         certezas[imgdetectar[i]]=[np.mean(lista), np.std(lista)]
302         lista1 += lista
303     return (certezas, np.mean(lista1), np.std(lista1))
304
305
306
307 def dibujoRectangulos(dir_detect,tipo_det,dir_destino,etiquetasDict,
    detectadosDict):
308     """
309     Dibuja en cada imagen los rectángulos etiquetados y detectados
310     """
311     for key,value in etiquetasDict.items():
312         image=cv2.imread(os.path.join(dir_detect,tipo_det,"images",key))
313         for j in range(len(value)):
314             min=(value[j][0],value[j][1])
315             max=(value[j][2],value[j][3])
316             cv2.rectangle(image,min,max,color=(0,255,0),thickness=3)
317             cv2.imwrite(os.path.join(dir_destino,"comparacion-" + key),
    image)
318     for key,value in detectadosDict.items():
319         image=cv2.imread(os.path.join(dir_destino,"comparacion-" + key))
```

```
320         for j in range(len(value)):
321             try:
322                 min=(value[j][0][0],value[j][0][1])
323                 max=(value[j][0][2],value[j][0][3])
324                 cv2.rectangle(image,min,max,color=(255,0,0),thickness=3)
325                 cv2.imwrite(os.path.join(dir_destino,"comparacion-" + key),
    image)
326             except:
327                 continue
328 
329 
330 ############################################################
331 
332 diccionarioEtiquetas = etiquetas(dir_detect = PATH_DETECTAR, tipo_det =
    TIPO_DETECCION)
333 
334 print("Diccionario racimos etiquetados: \n")
335 pprint.pprint(diccionarioEtiquetas)
336 
337 print("\n","=" * int(ANCHO),"\n")
338 
339 
340 
341 #####
342 diccionarioQueDetecto = detectados(dir_detect = PATH_DETECTAR, tipo_det =
    TIPO_DETECCION, archivo_ans = NOMBRE_ARCHIVO)
343 
344 print("Diccionario con todas las detecciones: \n")
345 pprint.pprint(diccionarioQueDetecto)
346 
347 print("\n","=" * int(ANCHO),"\n")
348 
349 
350 
351 #####
352 diccionarioComparaciones = comparaciones(dir_detect = PATH_DETECTAR,
    tipo_det = TIPO_DETECCION, etiquetasDict = diccionarioEtiquetas,
    detectadosDict = diccionarioQueDetecto, porcentaje_interseccion_minima
    = str(PORCENTAJE_INTERSECCION))
353 
354 print("Diccionario con las comparaciones: \n")
355 pprint.pprint(diccionarioComparaciones)
356 
357 print("\n","=" * int(ANCHO),"\n")
358 
359 
360 
361 #####
```

```
(totalOtrasDetecciones, diccionarioOtrasDetecciones) = otrasDet(dir_detect = PATH_DETECTAR, tipo_det = TIPO_DETECCION, etiquetasDict = diccionarioEtiquetas, comparaciones = diccionarioComparaciones)

print("Diccionario Otras detecciones: \n")
pprint.pprint(diccionarioOtrasDetecciones)

print("\n","=" * int(ANCHO),"\n")



#####
print("Diccionario detecciones que se corresponden con racimos: \n")
deteccionesPosiblesRacimos={}
img=701
for i in range(100):
    deteccionesPosiblesRacimos["bunch0"+str(img)+".jpg"]=[]
    if not "bunch0"+str(img)+".jpg" in diccionarioQueDetecto.keys() - diccionarioOtrasDetecciones:
        for j in range(len(diccionarioQueDetecto["bunch0"+str(img)+".jpg"])):
            if not diccionarioQueDetecto["bunch0"+str(img)+".jpg"][j] in diccionarioOtrasDetecciones["bunch0"+str(img)+".jpg"]:
                deteccionesPosiblesRacimos["bunch0"+str(img)+".jpg"].append(diccionarioQueDetecto["bunch0"+str(img)+".jpg"][j])
    else:
        for j in range(len(diccionarioQueDetecto["bunch0"+str(img)+".jpg"])):
            deteccionesPosiblesRacimos["bunch0"+str(img)+".jpg"].append(diccionarioQueDetecto["bunch0"+str(img)+".jpg"][j])
    img=img+1
pprint.pprint(deteccionesPosiblesRacimos)

print("\n","=" * int(ANCHO),"\n")



#####
(totalEtiquetas, totalNoDetectados, diccionarioCuantosNoDetecto, deteccionesIncorrectas, deteccionesCorrectas) = sumas(dir_detect = PATH_DETECTAR, tipo_det = TIPO_DETECCION, etiquetasDict = diccionarioEtiquetas, comparaciones = diccionarioComparaciones, suma_areas_min = str(SUMA_AREAS), bien_det = deteccionesPosiblesRacimos)

print("Diccionario detecciones correctas de racimos: \n")
pprint.pprint(deteccionesCorrectas)

print("\n","=" * int(150),"\n")
```

```
398
399 print("Diccionario detecciones racimos no satifactorias: \n")
400 pprint.pprint(deteccionesIncorrectas)
401
402 print("\n","=" * int(150),"\n")
403
404 print("Diccionario cuántos racimos NO detecto: \n")
405 pprint.pprint(diccionarioCuantosNoDetecto)
406
407 print("\n","=" * int(ANCHO),"\n")
408
409
410
411 #####
412 (diccionarioMeanStdCertezas, meanCertezas, stdCertezas) = promedios(
        dir_detect = PATH_DETECTAR, tipo_det = TIPO_DETECCION, detectadosDict =
         diccionarioQueDetecto, comparaciones = diccionarioComparaciones)
413
414 print("Promedio (primer valor) y desviación estandar (segundo valor) de los
         niveles de certeza de los racimos detectados que se corresponden con
        un racimo etiquetado, POR IMAGEN: \n")
415 pprint.pprint(diccionarioMeanStdCertezas)
416
417 print("\n","=" * int(ANCHO),"\n")
418
419
420
421 #####
422 dibujoRectangulos(dir_detect = PATH_DETECTAR, tipo_det = TIPO_DETECCION,
        dir_destino = DESTINO, etiquetasDict = diccionarioEtiquetas,
        detectadosDict = diccionarioQueDetecto)
423
424
425
426 #####
427 sumaTotalDetecciones=0
428 sumaTotalCorrectos=0
429 sumaTotalIncorrectos=0
430 img=701
431 for i in range(100):
432     sumaTotalDetecciones += len(diccionarioQueDetecto["bunch0"+str(img)+".
        jpg"])
433     sumaTotalCorrectos += len(deteccionesCorrectas["bunch0"+str(img)+".jpg"
        ])
434     if not(deteccionesIncorrectas["bunch0"+str(img)+".jpg"]==[]):
435         sumaTotalIncorrectos += len(deteccionesIncorrectas["bunch0"+str(img
        )+".jpg"])
436     img+=1
437
```

```
438
439 print("Total racimos etiquetados = ", totalEtiquetas)
440 print("Total racimos detectados correctamente = ", totalEtiquetas -
    totalNoDetectados)
441 print("Total racimos NO detectados = ", totalNoDetectados)
442 print("Exhaustividad = ", ((totalEtiquetas - totalNoDetectados) /
    totalEtiquetas) * 100)
443
444 print("Total Detecciones del modelo = ", sumaTotalDetecciones)
445 print("Detecciones racimos satisfactorias = ", sumaTotalCorrectos)
446 print("Deteciones racimos NO satisfactorias = ", sumaTotalIncorrectos)
447 print("Total otras detecciones = ", totalOtrasDetecciones)
448 print("Total otras detecciones = ", sumaTotalDetecciones - (
    sumaTotalCorrectos + sumaTotalIncorrectos))
449 print("Precisión = ", (sumaTotalCorrectos / sumaTotalDetecciones) * 100)
450
451 print("Promedio del TOTAL de los niveles de certeza de los racimos
    detectados que se corresponden con un racimo etiquetado = ",
    meanCertezas)
452 print("Desviacion estandar del TOTAL de los niveles de certeza de los
    racimos detectados que se corresponden con un racimo etiquetado = ",
    stdCertezas)
```

Código C.1. Script de Python para analizar los resultados obtenidos.

Bibliografía

[1] Warren S. McCulloch y Walter Pitts. «A logical calculus of the ideas immanent in nervous activity». En: *The bulletin of mathematical biophysics* 5.4 (1943), págs. 115-133 (pág. 23).

[2] Claude E. Shannon. «A mathematical theory of communication». En: *The Bell system technical journal* 27.3 (1948), págs. 379-423 (pág. 19).

[3] Donald Olding Hebb. «The organization of behavior; a neuropsycholocigal theory». En: *A Wiley Book in Clinical Psychology* 62 (1949), pág. 78 (pág. 30).

[4] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton (Project Para)*. Cornell Aeronautical Laboratory, 1957 (pág. 23).

[5] Claude E. Shannon y Warren Weaver. *The Mathematical Theory of Communication*. The University of Illinois Press. Urbana, 1964 (pág. 17).

[6] John E. Dennis Jr. *RB Schnabel Numerical methods for unconstrained optimization and nonlinear equations*. 1983 (pág. 32).

[7] David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams. *Learning internal representations by error propagation*. Inf. téc. California Univ San Diego La Jolla Inst for Cognitive Science, 1985 (pág. 33).

[8] John McCarthy y Ed Feigenbaum. «Arthur Samuel: Pioneer in Machine Learning». En: *AI Magazine* 11.3 (1990), págs. 10-11 (pág. 8).

[9] Richard L. Burden, J. Douglas Faires, Rafael Iriarte Balderrama y col. *Análisis numérico*. 1996 (pág. 51).

[10] Hidefumi Kobatake y Yukiyasu Yoshinaga. «Detection of spicules on mammogram based on skeleton analysis». En: *IEEE Transactions on Medical Imaging* 15.3 (1996), págs. 235-245 (pág. 4).

[11] Yann LeCun y col. «Gradient-based learning applied to document recognition». En: *Proceedings of the IEEE* 86.11 (1998), págs. 2278-2324 (págs. 3, 38).

[12] K-K Sung y Tomaso Poggio. «Example-based learning for view-based human face detection». En: *IEEE Transactions on pattern analysis and machine intelligence* 20.1 (1998), págs. 39-51 (pág. 4).

[13] Jerrold Eldon Marsden y Anthony Tromba. *Vector calculus*. Macmillan, 2003 (pág. 30).

[14] David Jacobs. «Correlation and convolution». En: *Class Notes for CMSC* 426 (2005) (págs. 39-41).

[15] Charles G. Morris, Albert A. Maisto y María Elena Ortiz Salinas. *Introducción a la Psicología*. Pearson Educación, 2005 (pág. 23).

[16] Tom Michael Mitchell. *The discipline of machine learning*. Vol. 9. Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2006 (pág. 8).

[17] Travis E. Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006 (pág. 78).

[18] David Kriesel. *A Brief Introduction to Neural Networks*. 2007. URL: `http://www.dkriesel.com` (págs. 9, 23, 25-27).

[19] Ignacio Serra, Ricardo Merino y Marcela Hidalgo. «Sistemas de conducción en vid. Incidencia en la producción y calidad del vino: una revisión». En: *Agro-Ciencia* 25.1 (2009), págs. 41-48 (pág. 70).

[20] Mark Everingham y col. «The pascal visual object classes (voc) challenge». En: *International journal of computer vision* 88.2 (2010), págs. 303-338 (pág. 52).

[21] Xavier Glorot y Yoshua Bengio. «Understanding the difficulty of training deep feedforward neural networks». En: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, págs. 249-256 (pág. 35).

[22] Piotr Dollar y col. «Pedestrian detection: An evaluation of the state of the art». En: *IEEE transactions on pattern analysis and machine intelligence* 34.4 (2011), págs. 743-761 (pág. 4).

[23] Fabian Pedregosa y col. «Scikit-learn: Machine learning in Python». En: *Journal of machine Learning Research* 12 (2011), págs. 2825-2830 (pág. 2).

[24] Yaser S. Abu-Mostafa, Malik Magdon-Ismail y Hsuan-Tien Lin. *Learning from data*. Vol. 4. AMLBook New York, NY, USA, 2012 (pág. 12).

[25] Léon Bottou. «Stochastic gradient descent tricks». En: *Neural networks: Tricks of the trade*. Springer, 2012, págs. 421-436 (págs. 31-33).

[26] Ronald E. Walpole y col. *Probabilidad y estadística para ingeniería y ciencias*. Pearson, 2012 (págs. 16, 17, 19, 20).

[27] Kruse R. y col. *Computational Intelligence: A Methodological Introduction*. Texts in Computer Science. Springer, 2013 (pág. 23).

[28] D. Font y col. «Counting red grapes in vineyards by detecting specular spherical reflection peaks in RGB images obtained at night with artificial illumination». En: *Computers and electronics in agriculture* 108 (2014), págs. 105-111 (pág. 4).

[29] Ross Girshick y col. «Rich feature hierarchies for accurate object detection and semantic segmentation». En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, págs. 580-587 (pág. 4).

[30] Shai Shalev-Shwartz y Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge University Press, 2014 (págs. 8, 12, 31).

[31] Chenyi Chen y col. «Deepdriving: Learning affordance for direct perception in autonomous driving». En: *Proceedings of the IEEE international conference on computer vision*. 2015, págs. 2722-2730 (pág. 4).

[32] Ross Girshick. «Fast R-CNN». En: *Proceedings of the IEEE international conference on computer vision*. 2015, págs. 1440-1448 (pág. 4).

[33] Sergey Ioffe y Christian Szegedy. «Batch normalization: Accelerating deep network training by reducing internal covariate shift». En: *arXiv preprint arXiv:1502.03167* (2015) (pág. 35).

[34] Y. LeCun, Y. Bengio y G. Hinton. «Deep learning». En: *Nature* 521 (2015), págs. 436-444. DOI: 10.1038/nature14539 (págs. 2, 3).

[35] Sunil Ray. *7 Regression Techniques you should know*. Visitado en Enero 2021. 2015. URL: https://www.analyticsvidhya.com/blog/2015/08/comprehensive-guide-regression/ (pág. 16).

[36] Tzutalin. *LabelImg. Git code*. 2015. URL: https://github.com/tzutalin/labelImg (pág. 70).

[37] Vladimir N. Vapnik y A. Ya. Chervonenkis. «On the uniform convergence of relative frequencies of events to their probabilities». En: *Measures of complexity*. Springer, 2015, págs. 11-30 (pág. 31).

[38] Vincent Dumoulin y Francesco Visin. «A guide to convolution arithmetic for deep learning». En: *arXiv preprint arXiv:1603.07285* (2016) (pág. 41).

[39] Kaiming He y col. «Deep residual learning for image recognition». En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, págs. 770-778 (págs. 36, 37, 47).

[40] Wei Liu y col. «SSD: Single Shot MultiBox Detector». En: *European conference on computer vision*. Springer. 2016, págs. 21-37 (pág. 49).

[41] Joseph Redmon y col. «You Only Look Once: Unified, real-time object detection». En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, págs. 779-788 (págs. 4, 49, 57, 65, 76).

[42] Shaoqing Ren y col. «Faster R-CNN: towards real-time object detection with region proposal networks». En: *IEEE transactions on pattern analysis and machine intelligence* 39.6 (2016), págs. 1137-1149 (págs. 4, 49).

[43] Mark Schmidt. *Argmax and Max Calculus*. https://www.cs.ubc.ca/~schmidtm/Courses/Notes/max.pdf. Visitado en Enero 2021. 2016 (pág. 13).

[44] Zhenheng Yang y Ramakant Nevatia. «A multi-scale cascade fully convolutional network face detector». En: *2016 23rd International Conference on Pattern Recognition (ICPR)*. IEEE. 2016, págs. 633-638 (pág. 4).

[45] F. Abdelghafour y col. «Potential of on-board colour imaging for in-field detection and counting of grape bunches at early fruiting stages». En: *Advances in Animal Biosciences* 8.2 (2017), pág. 505 (pág. 4).

[46] Arturo Aquino y col. «A new methodology for estimating the grapevine-berry number per cluster using image analysis». En: *Biosystems Engineering* 156 (abr. de 2017), págs. 80-95. DOI: 10.1016/j.biosystemseng.2016.12.011 (pág. 4).

[47] Suchet Bargoti y James Underwood. «Deep fruit detection in orchards». En: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, págs. 3626-3633 (pág. 4).

[48] Kaiming He y col. «Mask R-CNN». En: *Proceedings of the IEEE international conference on computer vision*. 2017, págs. 2961-2969 (pág. 49).

[49] M. Tim Jones. *Models for machine learning*. Visitado en Enero 2021. 2017. URL: https://developer.ibm.com/articles/cc-models-machine-learning/ (págs. 9, 11).

[50] Andrej Karpathy y Fei-Fei Li. *CS231n: Stanford course on Convolutional Neural Networks for Visual Recognition*. Visitado en Enero 2021. 2017. URL: https://cs231n.github.io/ (págs. 40, 42, 43, 45).

[51] Hung Le y Ali Borji. «What are the receptive, effective receptive, and projective fields of neurons in convolutional neural networks?» En: *arXiv preprint arXiv:1705.07049* (2017) (pág. 42).

[52] Tsung-Yi Lin y col. «Focal loss for dense object detection». En: *Proceedings of the IEEE international conference on computer vision*. 2017, págs. 2980-2988 (pág. 49).

[53] Joseph Redmon y Ali Farhadi. «YOLO9000: better, faster, stronger». En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, págs. 7263-7271 (pág. 57).

[54] Arturo Aquino y col. «Automated early yield prediction in vineyards from on-the-go image acquisition». En: *Computers and Electronics in Agriculture* 144 (ene. de 2018), págs. 26-36. DOI: 10.1016/j.compag.2017.11.026 (pág. 4).

[55] Arturo Aquino y col. «vitisBerry: An Android-smartphone application to early evaluate the number of grapevine berries by means of image analysis». En: *Computers and Electronics in Agriculture* 148 (2018), págs. 19-28 (pág. 4).

[56] Luca Coviello. «Deep neural network and precision agriculture for grape yield estimation». 2018. URL: http://oa.upm.es/56667/ (págs. 4, 73).

[57] Raúl Gómez. *Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names*. Visitado en Enero 2021. 2018. URL: https://gombru.github.io/2018/05/23/cross_entropy_loss/ (págs. 17, 19, 20).

[58] Luis Gonzales. *YOLOv3 loss function*. Visitado en Marzo 2021. 2018. URL: https://stats.stackexchange.com/questions/380012/yolov3-loss-function (pág. 65).

[59] Carol Hsin. *Yolo Object Detectors: Final Layers and Loss Functions*. Visitado en Marzo 2021. 2018. URL: https://medium.com/oracledevs/final-layers-and-loss-functions-of-single-stage-detectors-part-1-4abbfa9aa71c (pág. 65).

[60] Jonathan Hui. *mAP (mean Average Precision) for Object Detection*. Visitado en Enero 2021. 2018. URL: https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173 (pág. 52).

[61] Ayoosh Kathuria. *What's new in YOLO v3?* Visitado en Diciembre 2020. 2018. URL: https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b (pág. 65).

[62] Mohammed A. Al-Masni y col. «Simultaneous detection and classification of breast masses in digital mammograms via a deep learning YOLO-based CAD system». En: *Computer methods and programs in biomedicine* 157 (2018), págs. 85-94 (pág. 4).

[63] Agnieszka Mikolajczyk y Michal Grochowski. «Data augmentation for improving deep learning in image classification problem». En: *2018 International Interdisciplinary PhD Workshop (IIPhDW)*. IEEE, mayo de 2018. DOI: 10.1109/iiphdw.2018.8388338 (pág. 72).

[64] Moses y John Olafenwa. *ImageAI, an open source python library built to empower developers to build applications and systems with self-contained Computer Vision capabilities*. Mar. de 2018. URL: https://github.com/OlafenwaMoses/ImageAI (pág. 76).

[65] Arthur Ouaknine. *Review of Deep Learning Algorithms for Object Detection*. Visitado en Febrero 2021. 2018. URL: https://medium.com/zylapp/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852 (págs. 48, 49).

[66] Joseph Redmon. *yolov3.cfg*. Visitado en Enero 2021. 2018. URL: https://github.com/pjreddie/darknet/blob/master/cfg/yolov3.cfg (págs. 58-61).

[67] Joseph Redmon y Ali Farhadi. «Yolov3: An incremental improvement». En: *arXiv preprint arXiv:1804.02767* (2018) (págs. 49, 57-61, 64-66, 76).

[68] Google Research. *Featured prediction Competition. Google AI Open Images - Object Detection Track*. Visitado en Febrero 2021. 2018. URL: https://www.kaggle.com/c/google-ai-open-images-object-detection-track/ (pág. 52).

[69] Fábio Franco Uechi. *YOLO v3 Layers*. Visitado en Enero 2021. 2018. URL: https://gist.github.com/fabito/a49bb6a5593594f26275bc90baba6e32 (págs. 59-61).

[70] *YOLOv3 Model Defined in Keras*. Visitado en Enero 2021. 2018. URL: https://github.com/qqwweee/keras-yolo3/blob/master/yolo3/model.py (pág. 65).

[71] Rokas Balsys. *YOLO v3 theory explained*. Visitado en Enero 2021. 2019. URL: https://pylessons.medium.com/yolo-v3-theory-explained-33100f6d193 (págs. 63, 65).

[72] Jason Brownlee. *Overfitting and Underfitting With Machine Learning Algorithms*. Visitado en Enero 2021. 2019. URL: https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/ (pág. 14).

[73] Jason Brownlee. *What is the Difference Between a Parameter and a Hyperparameter?* Visitado en Enero 2021. 2019. URL: https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/ (pág. 9).

[74] Zhaowei Cai y Nuno Vasconcelos. «Cascade R-CNN: high quality object detection and instance segmentation». En: *IEEE transactions on pattern analysis and machine intelligence* (2019) (pág. 49).

[75] Luca Coviello y col. «In-field grape berries counting for yield estimation using dilated CNNs». En: *arXiv preprint arXiv:1909.12083* (2019) (pág. 4).

[76] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019 (págs. 2, 8, 9, 13, 14, 16, 17, 19-22, 24-26, 35, 36, 40, 42-46, 64).

[77] Hugegene. *Calculating Loss of Yolo (v3) Layer*. Visitado en Marzo 2021. 2019. URL: https://towardsdatascience.com/calculating-loss-of-yolo-v3-layer-8878bfaaf1ff (pág. 65).

[78] Anand Koirala y col. «Deep learning for real-time fruit detection and orchard fruit load estimation: Benchmarking of 'MangoYOLO'». En: *Precision Agriculture* 20.6 (2019), págs. 1107-1135 (pág. 4).

[79] Ethan Yanjia Li. *Dive Really Deep into YOLO v3: A Beginner's Guide*. Visitado en Enero 2021. 2019. URL: https://towardsdatascience.com/dive-really-deep-into-yolo-v3-a-beginners-guide-9e3d2666280e (págs. 58, 60, 63, 65, 66).

[80] Pankaj Mehta y col. «A high-bias, low-variance introduction to machine learning for physicists». En: *Physics reports* 810 (2019), págs. 1-124 (págs. 12-14, 28, 31-33, 35, 36, 40, 43, 46).

[81] Marisa Mohr. *The Mystery of Entropy: How to Measure Unpredictability in Machine Learning*. Visitado en Enero 2021. 2019. URL: https://www.inovex.de/blog/the-mystery-of-entropy-how-to-measure-unpredictability-in-machine-learning/ (págs. 19, 20).

[82] Google Research. *Research prediction Competition. Open Images 2019 - Object Detection*. Visitado en Febrero 2021. 2019. URL: https://www.kaggle.com/c/open-images-2019-object-detection/ (pág. 52).

[83] Pulkit Sharma. *Image Classification vs. Object Detection vs. Image Segmentation*. Visitado en Febrero 2021. 2019. URL: https://medium.com/analytics-vidhya/image-classification-vs-object-detection-vs-image-segmentation-f36db85fe81 (pág. 48).

[84] Connor Shorten y Taghi M. Khoshgoftaar. «A survey on Image Data Augmentation for Deep Learning». En: *Journal of Big Data* 6.1 (jul. de 2019). DOI: 10.1186/s40537-019-0197-0 (págs. 72, 73).

[85] Yunong Tian y col. «Apple detection during different growth stages in orchards using the improved YOLO-V3 model». En: *Computers and Electronics in Agriculture* 157 (feb. de 2019), págs. 417-426. DOI: 10.1016/j.compag.2019.01.012 (pág. 4).

[86] Laura Zabawa y col. «Detection of single grapevine berries in images using fully convolutional neural networks». En: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 2019 (págs. 5, 73).

[87] Zhong-Qiu Zhao y col. «Object detection with deep learning: A review». En: *IEEE transactions on neural networks and learning systems* 30.11 (2019), págs. 3212-3232 (págs. 4, 49).

[88] Zhengxia Zou y col. «Object detection in 20 years: A survey». En: *arXiv preprint arXiv:1905.05055* (2019) (pág. 49).

[89] Uri Almog. *YOLO V3 Explained*. Visitado en Enero 2021. 2020. URL: https://towardsdatascience.com/yolo-v3-explained-ff5b850390f (págs. 65, 66).

[90] Alexey Bochkovskiy, Chien-Yao Wang y Hong-Yuan Mark Liao. «Yolov4: Optimal speed and accuracy of object detection». En: *arXiv preprint arXiv:2004.10934* (2020) (pág. 57).

[91] Fernando Sancho Caparrini. *Entrenamiento de Redes Neuronales: mejorando el Gradiente Descendente*. Visitado en Enero 2021. 2020. URL: http://www.cs.us.es/~fsancho/?e=165 (pág. 26).

[92] Marc Peter Deisenroth, A. Aldo Faisal y Cheng Soon Ong. *Mathematics for machine learning*. Cambridge University Press, 2020 (pág. 16).

[93] *Detailed explanation of YOLOv3 loss function*. Visitado en Marzo 2021. 2020. URL: https://www.fatalerrors.org/a/detailed-explanation-of-yolov3-loss-function.html (pág. 63).

[94] ImageNet. *Research prediction Competition. ImageNet Object Localization Challenge*. Visitado en Febrero 2021. 2020. URL: https://www.kaggle.com/c/imagenet-object-localization-challenge/ (pág. 52).

[95] Shivan Kumar. *Underfitting and Overfitting*. Visitado en Enero 2021. 2020. URL: https://www.kaggle.com/getting-started/183376 (pág. 15).

[96] Li Liu y col. «Deep learning for generic object detection: A survey». En: *International journal of computer vision* 128.2 (2020), págs. 261-318 (pág. 49).

[97] Rafael Padilla, Sergio L. Netto y Eduardo A. B. da Silva. «A Survey on Performance Metrics for Object-Detection Algorithms». En: *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*. IEEE. 2020, págs. 237-242 (págs. 49, 50, 52, 53, 55, 56).

[98] Synced. *YOLO Creator Joseph Redmon Stopped CV Research Due to Ethical Concerns*. Visitado en Marzo 2021. 2020. URL: https://syncedreview.com/2020/02/24/yolo-creator-says-he-stopped-cv-research-due-to-ethical-concerns/ (pág. 57).

[99] The ImageMagick Development Team. *ImageMagick*. Ver. 7.0.10. 12 de jun. de 2020. URL: https://imagemagick.org (pág. 73).

[100] Chien-Yao Wang, Alexey Bochkovskiy y Hong-Yuan Mark Liao. «Scaled-YOLOv4: Scaling Cross Stage Partial Network». En: *arXiv preprint arXiv:2011.08036* (2020) (pág. 57).

[101] Aston Zhang y col. *Dive into Deep Learning*. https://d2l.ai. 2020 (pág. 40).

[102] Xuan Bi y col. «Tensors in Statistics». En: *Annual Review of Statistics and Its Application* 8.1 (2021). DOI: 10.1146/annurev-statistics-042720-020816 (págs. 38, 39).

[103] Glenn Jocher y col. *ultralytics/yolov5: v4.0 - nn.SiLU() activations, Weights & Biases logging, PyTorch Hub integration*. Ver. v4.0. Ene. de 2021. DOI: 10.5281/zenodo.4418161. URL: https://doi.org/10.5281/zenodo.4418161 (pág. 57).

[104] *Adaptive Histogram Equalization*. Visitado en Julio 2020. URL: https://imagemagick.org/script/clahe.php (pág. 73).

[105] *Annotated List of Command-line Options: -flop*. Visitado en Marzo 2020. URL: https://imagemagick.org/script/command-line-options.php#flop (pág. 73).

[106] *Annotated List of Command-line Options: -solarize* percent-threshold. Visitado en Junio 2020. URL: https://imagemagick.org/script/command-line-options.php#solarize (pág. 73).

[107] *Backpropagation and SGD*. Visitado en Enero 2021. URL: https://denizyuret.github.io/Knet.jl/latest/backprop/ (pág. 33).

[108] *Bases biológicas del comportamiento*. Visitado en Enero 2021. URL: https://filosert.wordpress.com/temas/4-bases-biologicas-del-comportamiento/ (pág. 24).

[109] *COCO: Common Objects in Context*. Visitado en Febrero 2021. URL: https://cocodataset.org/ (pág. 52).

[110] *COCO: Common Objects in Context. Detection Evaluation*. Visitado en Enero 2021. URL: https://cocodataset.org/#detection-eval (pág. 53).

[111] *ImageMagick v6 Examples – Blurring and Sharpening Images*. Visitado en Febrero 2020. URL: https://legacy.imagemagick.org/Usage/blur/ (pág. 74).

[112] *ImageMagick v6 Examples – Color Modifications: Solarize Coloring*. Visitado en Junio 2020. URL: https://legacy.imagemagick.org/Usage/color_mods/#solarize (pág. 73).

[113] *ImageMagick v6 Examples – Image Transformations: Oil Painting, blobs of color*. Visitado en Febrero 2020. URL: https://legacy.imagemagick.org/Usage/transform/#paint (pág. 74).

[114] *Information Content*. Visitado en Enero 2021. URL: https://en.wikipedia.org/wiki/Information_content (pág. 18).

[115] *Noise*. Visitado en Febrero 2020. URL: https://im.snibgo.com/noise.htm (pág. 74).

[116] Peter Sadowski. *Notes on Backpropagation*. https://www.ics.uci.edu/~pjsadows/notes.pdf. Visitado en Enero 2021 (pág. 20).