UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Detección de enfermedades y plagas en cultivos mediante Machine Learning
AUTORES: Pereyra, Mauro Ezequiel
DIRECTOR: Urbieta, Matias
CODIRECTOR:
ASESOR PROFESIONAL:
CARRERA: Licenciatura en Sistemas

Resumen

Las enfermedades y plagas en cultivos son una problemática que no sólo amenazan a la seguridad alimentaria mundial, sino que también generan grandes consecuencias económicas. Su detección temprana es un factor clave para conseguir erradicar o minimizar los perjuicios que pudiesen ocasionar. Considerando los avances en el Machine Learning (y en las Redes Neuronales en particular), la mejora en el hardware de los teléfonos móviles y el incremento en el acceso de la población a estos dispositivos, se propone el desarrollo de una plataforma cloud que posibilite la detección, en tiempo real, de enfermedades y plagas en cultivos a través de la cámara del teléfono celular. La plataforma mencionada se basa en la composición de soluciones cloud existentes y gratuitas, y se encuentra diseñada para ser configurable, extensible y fácil de utilizar. Ademas, puede ser aplicada a cualquier dominio de interés ya que es genérica, lo que le permite ser entrenada para detectar cualquier objeto deseado.

Palabras Clave

Machine Learning. Redes Neuronales Convolucionales. Transfer Learning. Detección de objetos. Detección de síntomas en cultivos. TensorFlow. Oidio. Cladosporium.

Trabajos Realizados

Inicialmente se investigó sobre las técnicas de Machine Learning utilizadas en la actualidad para la identificación de objetos en imágenes. Luego se analizaron diferentes investigaciones relacionadas a la detección de síntomas en cultivos mediante el uso de redes neuronales. A continuación, se diseñó y desarrolló la plataforma propuesta. Finalmente, se realizó el entrenamiento de la red utilizando imágenes de hojas de tomate con síntomas de oidio y cladosporium, el cual fue evaluado observando las diferentes métricas proporcionadas por TensorBoard y las predicciones realizadas por la aplicación móvil en tiempo real.

Conclusiones

La precisión de las predicciones realizadas sobre hojas con oidio y cladosporium demuestra que la plataforma puede ser utilizada en la agricultura como una herramienta adicional para la detección de enfermedades y plagas con el fin de combatir la problemática y de reducir sus consecuencias. Al ser una plataforma genérica, la misma puede ser utilizada en cualquier dominio de interés para detectar los objetos deseados. Al dia de hoy no hemos encontrado una plataforma similar a la propuesta en este trabajo, por lo que se espera que la misma haga una contribución significativa en las áreas en donde se la utilice y que a su vez, sea un recurso valioso de aprendizaje para todo aquel que desee iniciarse en el mundo del Machine Learning.

Trabajos Futuros

Debido a las características de la arquitectura, los trabajos futuros a ser aplicados sobre la misma son de lo más variados. En primer lugar se propone la realización de pruebas de campo, en donde se pueda verificar el funcionamiento de la misma. Además, se proponen ciertas mejoras técnicas, entre la que se encuentra la adaptación del código a TensorFlow 2, la posibilidad de entrenar redes neuronales propias y la ampliación, en cuanto a funcionalidad, de la aplicación movil.

Fecha de la presentación: Agosto 2020

Agradecimientos

A mis padres, los cuales son mi ejemplo a seguir.

A mis hermanos, por su apoyo constante. A Ale, por estar siempre a mi lado.

A mi compañera de vida, por su contención, ayuda y paciencia.

A Matías, por su acompañamiento permanente durante
el transcurso de este trabajo.

A la Facultad de Informática de la UNLP, directivos, docentes y
alumnos con los cuales recorrí este gran camino.

Índice general

Índice de figuras

1. Introducción

1.1.Motivación

La seguridad alimentaria refiere al acceso físico, social y económico de las personas a alimentos seguros, nutritivos y en cantidad suficiente como para satisfacer sus necesidades alimenticias y sus preferencias alimentarias [2].
Las enfermedades y plagas en cultivos constituyen una amenaza para ella, ya que pueden dañar los cultivos, lo que conlleva a una reducción en la disponibilidad y acceso a los alimentos causando un aumento en el precio de los mismos [3].

El informe "El estado de la seguridad alimentaria y la nutrición en el mundo" realizado en el año 2019 por la FAO (Organización de las Naciones Unidas para la Agricultura y la Alimentación), UNICEF y la OMS (Organización Mundial de la Salud) destaca que en el 2018 se encontraban 821 millones de personas con desnutrición crónica en el mundo, en comparación a las 811 millones del año anterior, lo cual corresponde a aproximadamente una de cada nueve personas en el mundo. El porcentaje de personas con padecimiento de hambre se encontraba en descenso desde hace unas décadas, hasta que en el año 2015 se produjo un quiebre en la tendencia y desde allí le continuaron tres años de constante aumento [4]. Tal situación demuestra que deben tomarse medidas para reducir la cantidad de personas que padecen hambre a nivel mundial. Más aún, la FAO sugiere que la producción agrícola deberá aumentar alrededor de un 70 % para el año 2050 para lograr alimentar a una población cada vez más numerosa [5].

Las plagas y enfermedades en los cultivos, además de atentar contra la erradicación del hambre, generan grandes consecuencias económicas. Se estima que ocasionan la pérdida de hasta el 40 % de los cultivos alimentarios a nivel mundial, lo cual conlleva a una pérdida monetaria por más de USD 220 mil millones cada año [6]. En el caso

de los pequeños productores, los cuales generan el 80 % de la producción mundial de alimentos [7, 8], las consecuencias económicas pueden llegar a ser devastadoras si no cuentan con los medios suficientes como para contrarrestar la situación.

Es crucial, entonces, el desarrollo e implementación de nuevas estrategias que permitan minimizar las consecuencias ocasionadas por esta problemática.

La detección temprana de las enfermedades y plagas es un factor clave para conseguir erradicar o minimizar los perjuicios que ello pueda ocasionar.

Aprovechando que esta problemática presenta generalmente síntomas visibles en los cultivos se puede hacer uso de las Redes Neuronales Convolucionales con el objetivo de entrenar modelos que posibiliten su detección. Además, teniendo en cuenta los avances tecnológicos relacionados al hardware de los teléfonos móviles se puede hacer uso de los mismos para, a través de sus cámaras de alta definición y de sus procesadores, obtener las imágenes y correr las predicciones sobre ellas en tiempo real en el dispositivo.

Las Redes Neuronales Convolucionales son una de las técnicas utilizadas hoy en día para tareas de detección de objetos en imágenes, debido en gran parte a su elevada precisión para realizar esa tarea [9]. Existe numerosa bibliografía sobre las mismas y se encuentran disponibles diferentes herramientas gratuitas, entre las que se encuentran TensorFlow [1] y TensorFlow Lite [2], los cuales permiten, entre otras cosas, el entrenamiento y uso de modelos de Machine Learning en dispositivos con recursos limitados, como los teléfonos móviles.

Considerando, además, que el porcentaje de la población que cuenta con un teléfono inteligente crece año tras año [10] y que no se han encontrado soluciones integrales que abarquen el proceso de entrenamiento y uso de modelos de Machine Learning capaces de detectar objetos en imágenes se propone el desarrollo de una plataforma cloud que, mediante el uso de Redes Neuronales Convolucionales, posibilite la detección en tiempo real de enfermedades y plagas en cultivos a través de la cámara del teléfono celular.

[1] https://www.tensorflow.org
[2] https://www.tensorflow.org/lite

1.2.Objetivo

El objetivo del presente trabajo es el desarrollo de una plataforma cloud de Machine Learning que posibilite, a través del entrenamiento de Redes Neuronales Convolucionales, la identificación de objetos en tiempo real a través de la cámara del teléfono celular.

La plataforma se basa en la composición de soluciones cloud existentes y gratuitas. Se busca que la misma resulte fácil de configurar, extender y modificar, permitiendo que cualquier persona que lo desee pueda hacer uso de ella.

Al ser genérica en lo que puede detectar, la plataforma puede ser utilizada en cualquier dominio de interés. Se podría utilizar en el campo de la medicina para la evaluación de lunares y de manchas en la piel, y para la identificación de anomalías en tomografías y radiografías. En cuestiones de seguridad y vigilancia, podría ser utilizada para monitorear el tránsito y para identificar personas y vehículos (a través de sus patentes). En el campo de la agricultura, la plataforma podría ser utilizada para evaluar la calidad de los cultivos, para el reconocimiento de especies y malezas, y para la realizar la identificación de enfermedades y plagas en cultivos, lo cual vamos a realizar en este trabajo, con motivo de proveer una alternativa para combatir la problemática y de servir de ayuda para la toma de decisiones en el agro.

1.3.Resultados esperados

De la realización de este trabajo, se esperan obtener los siguientes resultados:

1. El desarrollo de una metodología de trabajo que posibilite nuevas estrategias para la toma de decisiones en el agro utilizando Machine Learning.

2. Un estudio de las técnicas y herramientas disponibles en la actualidad para el entrenamiento y uso de redes neuronales que permitan la detección de objetos en imágenes.

3. Un estudio de las diferentes investigaciones relacionadas a Machine Learning y detección de síntomas en cultivos, para verificar las técnicas utilizadas, las estrategias seguidas y los resultados obtenidos.

4. El desarrollo e implementación de una plataforma tecnológica que entrene una red neuronal mediante el uso de diferentes servicios Cloud y que corra el modelo entrenado en un dispositivo móvil para permitir la detección de objetos en tiempo real y sin la necesidad de estar conectado a internet (salvo para la descarga del modelo y sus sucesivas actualizaciones).

5. La obtención de un recurso que simplifique el entrenamiento y uso de Redes Convolucionales y que sea de utilidad para el que desee aprender sobre Machine Learning en general y redes neuronales en particular.

2. Preliminares

2.1.Introducción

La plataforma de Machine Learning desarrollada para cumplir con los objetivos de este trabajo de investigación hace uso de Redes Neuronales Convolucionales con aprendizaje supervisado para entrenar un modelo capaz de detectar objetos en imágenes. Estas redes son un tipo de Red Neuronal Artificial y están diseñadas específicamente para trabajar con datos de entrada que poseen forma de arreglo (como imágenes) y para realizar tareas como clasificación de imágenes, segmentación y detección de objetos.

A continuación, procederemos a explicar las Redes Neuronales Convolucionales con un enfoque top-down. Primero se realizará una introducción al Machine Learning en general, luego se describirá los fundamentos y funcionamiento de las Redes Neuronales Artificiales y por último, se profundizará en las Redes Neuronales Convolucionales.

2.2.Machine Learning

El machine learning, o aprendizaje automático, es uno de los subcampos de la Inteligencia Artificial que, de acuerdo al pionero en el tema Arthur Samuel, proporciona a los sistemas la capacidad de aprender y mejorar automáticamente a partir de la experiencia sin la necesidad de ser programados explícitamente [11].

Podemos, además, considerar la siguiente definición formal dada por Tom Michel, en donde se refiere a los algoritmos estudiados en el campo de Machine Learning como "Se dice que un programa de computadora aprende de una experiencia E respecto a alguna tarea T y alguna medida de rendimiento P, si el rendimiento de sobre T medido por P mejora con la experiencia E." [12].

A diferencia de la programación tradicional, en donde se provee un conjunto de reglas (programa) y datos para ser procesados acorde con dichas reglas para así obtener una respuesta como salida del programa, con el Machine Learning, se brindan los datos como entrada al igual que las respuestas esperadas de dichos datos con el fin de obtener como salida las reglas que nos permiten hacer el mapeo efectivo entre las entradas y sus correspondientes salidas. Estas reglas pueden ser luego aplicadas a nuevos datos para producir respuestas generadas automáticamente por las reglas que el sistema "aprendió".

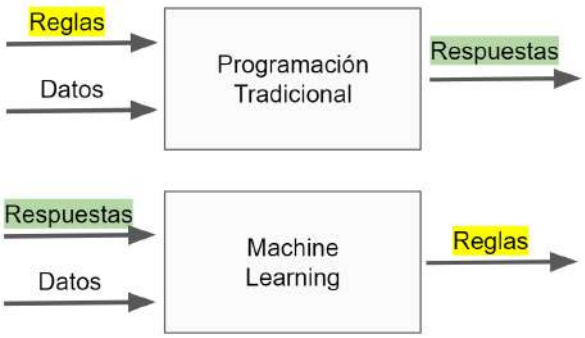


Figura 2.1: Programación tradicional versus Machine Learning

El objetivo de los algoritmos de Machine Learning es entonces el de construir un modelo que capture el conocimiento aprendido sobre los datos de entrada y que gracias a este modelo posteriormente pueda inferir conocimiento sobre nuevos datos. Para ello, se distinguen dos fases: la fase de aprendizaje y la fase de inferencia. La primera de las fases está orientada a la construcción de dicho modelo en base a datos de entrada (datos de entrenamiento). La segunda fase ocurre cuando se utiliza el modelo generado para inferir datos totalmente nuevos y desconocidos.

Se puede considerar al modelo mencionado como una función, que toma datos como entrada y devuelve un resultado o predicción sobre esos datos.

2.2.1. Tipos de Aprendizaje

Existen diferentes tipos de aprendizaje en Machine Learning, entre los que podemos destacar el aprendizaje supervisado, el no supervisado, el semi-supervisado y el aprendizaje por refuerzo, los cuales difieren en la forma en la que el algoritmo aprende.

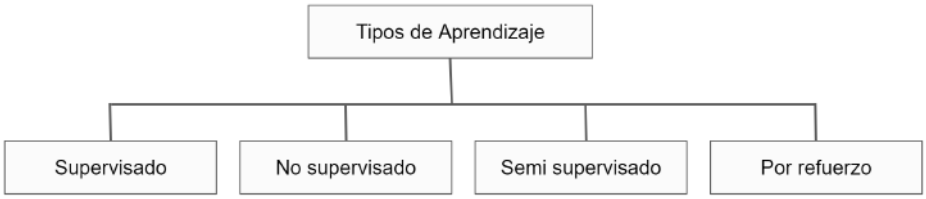


Figura 2.2: Tipos de aprendizaje en Machine Learning

Aprendizaje supervisado En el aprendizaje supervisado, los algoritmos trabajan con datos "etiquetados", intentado encontrar una función que, dadas las variables de entrada (datos de entrada), les asigne el resultado correcto. El algoritmo se entrena con un "histórico" de datos y así "aprende" patrones y reglas que le permiten inferir resultados sobre nuevos datos de entrada.

El aprendizaje supervisado se suele utilizar en:

- Problemas de clasificación, en donde el resultado de la predicción es una (o múltiples) etiquetas o clases discretas. Un ejemplo de ello es predecir si un cliente de una empresa es un potencial moroso (etiquetas: moroso - no moroso), o el típico problema de detección de correo spam, en donde el resultado de la predicción es "spam" o "no spam".
- Problemas de regresión, en donde el resultado de la predicción es un valor o cantidad continua. Un ejemplo de ello es predecir el precio de una vivienda, en donde las variables de entrada podrían ser la localidad en donde se encuentra, el tamaño de la vivienda, los servicios con los que cuenta el barrio, etc.

Aprendizaje no supervisado El aprendizaje no supervisado tiene lugar cuando no se dispone de datos "etiquetados" para el entrenamiento, ya que las etiquetas son difíciles de obtener, no hay conocimiento de los datos o el etiquetado es muy costoso. Sólo se conocen los datos de entrada, pero no existen datos de salida que correspondan a un determinado input. Por tanto, sólo se puede describir la estructura de los datos, para intentar encontrar algún tipo de organización que simplifique el análisis. Por ello, tienen un carácter exploratorio.

Aprendizaje semi supervisado Esta técnica, como su nombre lo refiere, es una combinación entre el aprendizaje supervisado y el aprendizaje no supervisado, por lo que requiere un conjunto de datos etiquetados y otro con datos no etiquetado. Lo mencionado es particularmente útil cuando el etiquetado de los datos es una tarea que requiere mucho tiempo o esfuerzo, por ejemplo.

El proceso de aprendizaje semi-supervisado es el siguiente:

1.Recopilar los datos que poseen resultados.

2.Recopilar los datos que no poseen resultados.

3. Construir un modelo que aprenda mediante aprendizaje supervisado los datos que tienen resultados.

4. Utilizar ese modelo para etiquetar automáticamente el resto de los datos (no etiquetados).

5. Construir un nuevo modelo que aprenda mediante aprendizaje supervisado los datos etiquetados inicialmente y los datos etiquetados automáticamente.

6.Utilizar el nuevo modelo con datos nuevos.

Aprendizaje por refuerzo Esta técnica busca optimizar el resultado de un problema por medio de la prueba y error. Todo problema de aprendizaje por refuerzo está

compuesto por un agente (algoritmo) y por su entorno. El agente puede ser definido como una entidad con capacidades de memoria y deducción, y su misión es entrenarse en el entorno hasta alcanzar un desempeño óptimo. Por otro lado, el entorno representa el problema a resolver, el contexto con el que interactúa el agente, es decir, su fuente de información, y está estructurado como una secuencia de alternativas.

El proceso de aprendizaje por refuerzo es el siguiente: en cada alternativa el agente realiza una acción, recibiendo a cambio una recompensa. A medida que acumula recompensas su conocimiento sobre el entorno va a ser mayor. El aprendizaje finaliza cuando el agente es capaz de encontrar una secuencia de acciones, de entre todas las posibles, que le reporta la mayor recompensa acumulada.

2.2.2.Tareas de Visión Artificial

En cuanto a las tareas de Visión Artificial que el Machine Learning puede llevar a cabo se encuentran las siguientes:

Figura 2.3: Tareas comunes de Visión Artificial

Clasificación El problema de clasificación de imágenes es la tarea de asignar una categoría (de un conjunto de categorías conocidas y finitas) a una imagen. Por ejemplo, un algoritmo de clasificación de imágenes podría estar diseñado para detectar si una imagen contiene un Perro o un Gato.

Detección de objetos La detección de objetos es una técnica que trata de distinguir objetos en una imagen. Si bien está relacionado con la clasificación, es más específico en lo que identifica, aplicando clasificación a distintos objetos en una imagen y usando

cuadros delimitadores (bounding boxes) con el objetivo de mostrar dónde se encuentra cada objeto en la imagen.

Segmentación A diferencia de la detección de objetos, en donde se crea un cuadro delimitador alrededor de cada objeto detectado, la segmentación proporciona el contorno exacto del objeto a través de una "máscara" en la imagen.

Existen dos tipos de segmentación, la segmentación semántica y la segmentación de instancia. En la primera se trata a múltiples objetos de la misma clase como una sola entidad agrupada. En la primer imagen de la Figura2.4por ejemplo, hay 2 entidades (persona y mesa).

En cambio, en la segmentación de instancia se tratan varios objetos de la misma clase como objetos individuales (o instancias) distintos. En la segunda imagen de la Figura2.4por ejemplo, hay 7 entidades (6 personas y una mesa).

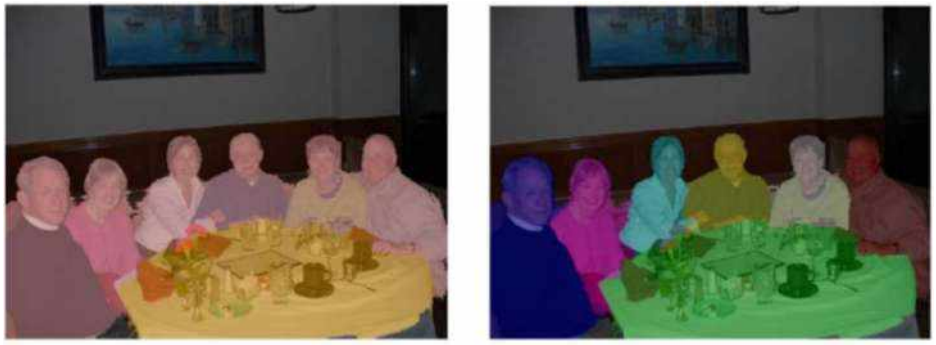


Figura 2.4: Tipos de segmentación

2.2.3.Evaluación de un modelo detector de objetos

Para evaluar la precisión de un modelo detector de objetos se suele hacer uso de la métrica conocida como intersección sobre unión (Intersection over Union - IoU).

Dado un rectángulo (conocido como *ground truth*) que representa la posición correcta de un objeto y un rectángulo que representa la predicción del modelo, la IoU se define como una división, en donde el numerador es el área de superposición entre el

cuadro delimitador predicho y el *ground truth* y el denominador es el área de unión comprendida tanto por el cuadro delimitador predicho como por el *ground truth*.

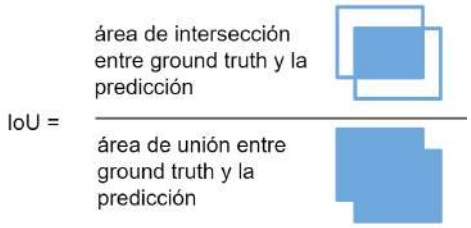


Figura 2.5: Definición de la métrica IOU

Hay que tener en cuenta que una coincidencia completa y total entre ambos cuadros delimitadores es poco realista y rara vez ocurre. Entonces, con motivo de poder evaluar si una detección es mala, buena o excelente se utiliza un valor mínimo de IoU (conocido como umbral, y cuyo valor es normalmente 0,5) desde el cual una predicción es correcta, como se observa en la Figura2.6.

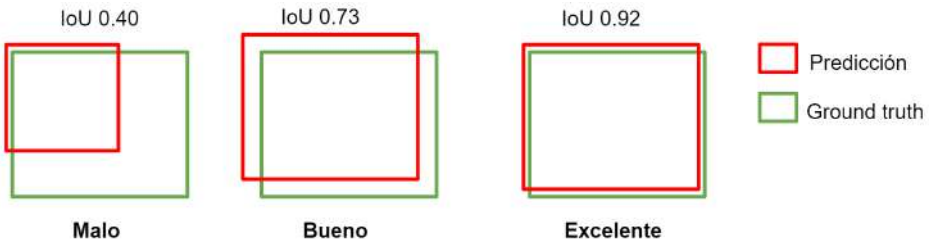


Figura 2.6: Ejemplo de uso de IoU, en donde IoU>0,5 es un buen resultado

Debido a que la IoU no tiene en cuenta la clase del objeto y solo se define para un solo objeto, en problemas de detección de objetos esta métrica no puede utilizarse directamente, ya que se quiere evaluar el modelo con respecto a un conjunto de prueba y teniendo en cuenta las diferentes clases de objetos. Por ello, se utiliza la métrica mAP (*mean Average Precision*), que se define a continuación.

Supongamos que queremos evaluar un detector de objetos en un conjunto de imágenes de prueba T y que en dichas imágenes pueden aparecer objetos de un conjunto de clases ζ. Comenzamos definiendo lo que es una detección correcta. Dado el *ground truth* de un objeto de una clase $C \in \zeta$, diremos que dicho objeto es detectado

correctamente si el modelo produce una predicción tal que la IoU del *ground truth* con el cuadro predicho es mayor a 0,5 y C es igual a la clase predicha.

Ahora calculamos el valor de IoU para cada cuadro del *ground truth*. Usando este valor y el umbral IoU establecido (0,5 por ejemplo), calculamos el número de detecciones correctas para cada clase en una imagen.

A continuación, para cada una de las imágenes, obtenemos la cantidad de objetos reales de una clase $C \in \zeta$ dada en esa imagen I; y definimos la precisión de la clase C en una imagen I de la siguiente manera:

$$Precision_{C,I} = \frac{\text{nº de detecciones correctas de la clase } C \text{ en } I}{\text{nº de objetos de la clase } C \text{ en } I}$$

Como estamos interesados en evaluar el modelo en el conjunto de imágenes T, definimos la *Average Precision* para la clase C como:

$$AveragePrecision_C = \frac{\sum\limits_{I \in T} Precision_{C,I}}{\text{nº total de imágenes de } T \text{ con objetos de la clase } C}$$

Para representar el rendimiento global del modelo realizaremos la media de todas las *Average Precision* de todas las clases que tengamos. Para ello se utiliza la métrica *mean Average Precision* (mAP), que se define como:

$$mAP = \frac{\sum\limits_{C \in \zeta} AveragePrecision_C}{\text{nº de clases}}$$

En algunos reportes se le añade al mAP el valor del umbral usado por la IoU, lo cual se denota como *mAP@0,5* por ejemplo.

Para concluir, otras métricas utilizadas también para evaluar la eficacia de un modelo es la precisión y el recall, que se definen como:

$$Precision = \frac{\text{TP}}{\text{FP} + \text{TP}}$$

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

En donde:

- True positive (TP): son los objetos detectados correctamente.
- False Positive (FP): son los objetos predichos por el modelo pero que no están en el *ground truth*.
- False Negative (FN): son los objetos que están en el *ground truth* pero que no han sido predichos.

2.2.4. Uso de Machine Learning en la detección de enfermedades y plagas en cultivos

El Machine Learning, y las Redes Neuronales Convolucionales en particular están siendo utilizadas desde ya hace varios años para distintas tareas en el campo de la Agricultura, entre las que se encuentra la detección de enfermedades en cultivos a través del análisis de sus síntomas. Existe una gran cantidad y variedad de investigaciones sobre ello, en donde se hace uso de diferentes técnicas y estrategias para lograr generar un modelo con una gran precisión.

En general las investigaciones relacionadas a las plantas utilizan el dataset de Plant Village [13] como fuente de imágenes, el cual contiene más de 50.000 imágenes de hojas sanas y hojas con síntomas de enfermedades que se encuentran divididas en 38 categorías por especie y por enfermedad. Este dataset fue utilizado, por ejemplo, para identificar 2 enfermedades en la banana con una precisión del 98.61 % mediante el uso de una arquitectura LeNet [14, 15]; para identificar 26 enfermedades en 14 cultivos con una precisión del 99.35 % [16] utilizando las arquitecturas AlexNet [9] y GoogleNet

[17]; para detectar la podredumbre negra en la manzana mediante el uso de Transfer Learning y de la arquitectura VGG-16 [18] con una precisión del 90.4 % [19]; y para detectar enfermedades en el cultivo de la papa con una precisión del 95 % [20].

Podemos citar además los siguientes trabajos:

Álvaro Fuentes et al. [21] hicieron uso de la arquitectura Faster R-CNN [22] con VGG-16 para el entrenamiento de un detector de enfermedades del tomate, el cual obtuvo una precisión del 83 % utilizando la técnica de aumento de imágenes.

Muhammad Waseem Tahir et al. [23] entrenaron un modelo capaz de detectar hongos con una precisión del 94.8 % utilizando un dataset propio con 40,800 imágenes etiquetadas.

Y Adhao Asmita Sarangdhar et al. [24] presentaron un sistema para la detección y control de cinco enfermedades en la hoja de algodón, el cual además realiza un monitoreo de la calidad del suelo. Una vez realizada la detección de la enfermedad, el nombre y su tratamiento se le proporciona a los agricultores mediante una aplicación móvil, la cual además ofrece distintos parámetros del suelo, como la humedad y temperatura, entre otros.

Si bien las investigaciones citadas han logrado buenos resultados en la identificación de las enfermedades, las mismas poseen ciertas limitaciones. La primera de ellas es el uso del dataset Plant Village como fuente de imágenes. Tal como describen algunos trabajos, el entrenamiento de redes utilizando ese dataset genera modelos con una precisión deficiente cuando deben inferir imágenes del mundo real. El motivo de ello es que Plant Village cuenta con imágenes tomadas en entornos controlados, o de laboratorio, en donde no se consideran situaciones reales, tal como la no uniformidad en el background de las imágenes, la diferente resolución de las cámaras, las diferencias en la iluminación, la variedad en el tamaño y formas de los síntomas y la posibilidad de la aparición de varias hojas en una misma imagen.

Otra de las limitaciones es que algunas de las investigaciones no permiten la detección de múltiples síntomas y enfermedades sobre una misma hoja, ya que realizan clasificación en vez de una detección de objetos, por lo que toman la hoja como un todo, en

vez de tomar los diferentes síntomas como entidades distintas.
Además, en los trabajos citados, el aumento de las imágenes no es un punto fuerte. En algunos de ellos directamente no se preparan ni aumentan las imágenes; en otros, se realiza únicamente una conversión a blanco y negro, pero en ninguno de ellos se implementa un aumento robusto, que contemple una gran variedad de transformaciones que permita generar diferentes condiciones para las imágenes.
Por ultimo, los trabajos analizados fueron desarrollados específicamente para trabajar sobre ciertas enfermedades y arquitecturas de redes neuronales. Mas aun, no mencionan la posibilidad de la utilización de los sistemas para la detección de otro tipo de objetos.

A diferencia de ellos, este trabajo propone una plataforma end-to-end que, entre otras características, permite entrenar un modelo capaz de detectar objetos en general y es totalmente configurable, en donde el aumento de las imágenes forma parte del flujo de trabajo y la elección de la arquitectura a utilizar dependerá del tipo de problema y de las preferencias del usuario.

2.3.Redes Neuronales Artificiales

Las redes neuronales artificiales (ANN) son una de las principales herramientas utilizadas en Machine Learning y se destacan por su eficiencia para encontrar patrones que resultan demasiado complejos o numerosos como para que un programador los extraiga y enseñe a la máquina a reconocer. Como sugiere la parte “neuronal” de su nombre, son sistemas inspirados en el cerebro humano que tienen la intención de replicar la forma en que nosotros aprendemos.

En las últimas décadas se han convertido en una parte esencial de la inteligencia artificial. Esto se debe, entre otros factores, a la aparición de la técnica denominada backpropagation (retropropagación), que permite a las redes ir ajustando sus parámetros internos en situaciones donde el resultado obtenido no coincide con lo que se espera, para así mejorar su precisión. Otro avance importante ha sido la aparición de las redes

neuronales convolucionales, en las que diferentes capas de una red multicapa extraen diferentes características con la finalidad de lograr reconocer lo que está buscando en una imagen, audio o texto. De estas redes vamos a hablar en la próxima sección.

Las Redes Neuronales Artificiales están basadas en el funcionamiento de las redes de neuronas biológicas. Están formadas por un conjunto de nodos conocidos como neuronas artificiales que están conectadas y transmiten señales entre sí, las cuales se comunican desde la entrada de la red hasta el final para generar una salida. Además, las neuronas de la red se encuentran agrupadas en distintas capas.

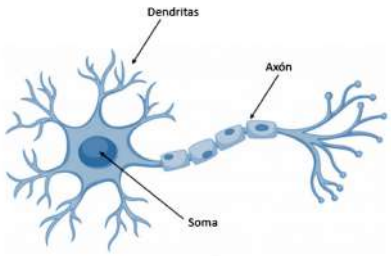


Figura 2.7: Estructura de una neurona biológica

Tal como se observa en la Figura2.7, las neuronas que todos tenemos en nuestro cerebro están compuestas por dendritas, por el soma y por el axón. Las dendritas se encargan de captar los impulsos nerviosos que emiten otras neuronas. Estos impulsos, se procesan en el soma y se transmiten a través del axón que emite un impulso nervioso hacia las neuronas contiguas.

Por su parte, a nivel esquemático, una neurona artificial se representa del siguiente modo:

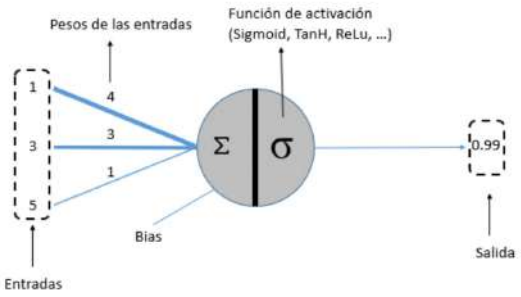


Figura 2.8: Neurona artificial

Para entender la Figura2.8, debemos tener presentes los siguientes conceptos:

- Peso (weight): Representa la fuerza/importancia de la conexión entre las neuronas de la red. El peso de la neurona artificial representa dos conceptos, la plasticidad y la fuerza de la señal. La plasticidad es el concepto que refiere a que el peso se va ajustando durante el entrenamiento de la red, y la fuerza de la señal es la cantidad de influencia que la neurona que se enciende tiene en las otras a las cuales está conectada de la siguiente capa.

- Bias (offset): Es una entrada adicional para las neuronas, su valor es siempre 1 y tiene su propio peso de conexión. Esto asegura que incluso cuando todas las entradas sean nulas (todas 0) habrá una activación en la neurona.

La neurona es la unidad básica de una red neuronal. Recibe cierto número de entradas y un bias, todos ellos con un peso asociado. La suma de sus entradas multiplicadas por los pesos asociados determina el "impulso nervioso" que recibe la neurona. Este valor se procesa en el interior de ella mediante una función de activación, la cual devuelve un valor que se envía como salida de la neurona.

En la figura2.9se puede observar una red neuronal con cuatro capas (1 capa de entrada, dos capas ocultas y una capa de salida):

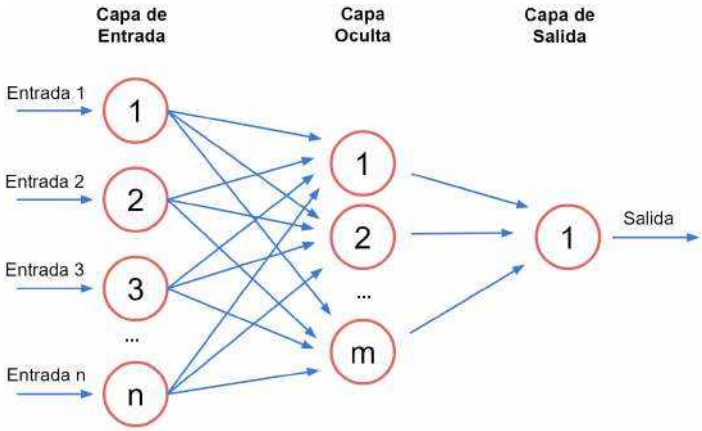


Figura 2.9: Red neuronal

La capa de entrada es la primera capa en la red neuronal. Recibe una serie de valores de entrada que envía a la siguiente capa para que continúen su camino por la red. No aplica ninguna operación en los valores de entrada y no tiene valores de pesos ni bias asociados.

La salida de la última capa es el resultado visible de la red (la predicción realizada), por lo que esta capa se conoce como capa de salida. La misma recibe los valores desde la última capa oculta.

Las capas que se sitúan entre la capa de entrada y la capa de salida se conocen como capas ocultas ya que desconocemos tanto los valores que contienen como las operaciones que se realizan en su interior, y además sus neuronas no son ni entradas ni salidas de la red. Estas capas poseen neuronas que aplican diferentes transformaciones a los datos de entrada y la última de las capas ocultas envía valores a la capa de salida.

La red que cuenta con más de una capa oculta es denominada deep neural network (red neuronal profunda).

El objetivo principal de este tipo de redes es aprender, modificándose automáticamente a sí mismas, para lograr realizar tareas complejas que no podrían ser realizadas mediante la clásica programación tradicional.

2.3.1.Funciones de activación

Anteriormente comentamos que en las neuronas se realizan transformaciones a los valores recibidos antes de ser enviados hacia la próxima neurona. Esas transformaciones son realizadas por las funciones de activación.

Las funciones de activación son utilizadas para convertir una entrada ilimitada en una salida con forma predecible. Además, sirven para introducir la no-linealidad en la red, lo que permite que el modelo cree conexiones complejas entre las entradas y las salidas de la red, lo cual es esencial para el entrenamiento de datos complejos como imágenes, audio y video.

Una función de activación comúnmente utilizada es la función *sigmoid* (sigmoide), la cual transforma los valores de entrada a un valor que se encuentra dentro del rango (0,1), en donde los valores positivos altos tienden de manera asintótica a 1 y los valores negativos bajos tienden de manera asintótica a 0.

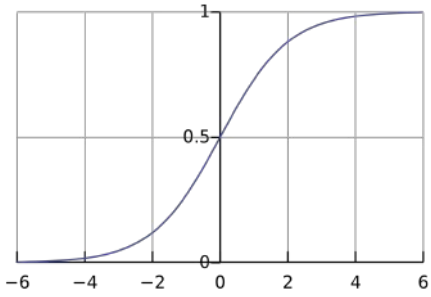


Figura 2.10: Función sigmoid

Otra función utilizada es *softmax*, que a diferencia de la función de activación anterior, se utiliza para la clasificación de varias clases. Esta función calcula la distribución de probabilidades de una clase entre ñ'clases diferentes, transformando las salidas a una representación en forma de probabilidades, de tal manera que la suma de todas ellas resulte igual a 1.

Por último, podemos mencionar a la función *Relu*, que transforma los valores introducidos anulando los valores negativos y dejando los positivos tal y como ingresan.

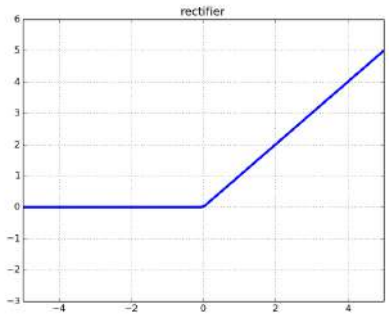


Figura 2.11: Función Relu

2.3.2.Entrenamiento de la Red Neuronal

Para lograr que una red neuronal realice la tarea deseada, es necesario entrenarla. El entrenamiento de una red neuronal se realiza modificando constantemente los valores de sus parámetros (pesos y bias) de manera de ir obteniendo progresivamente mejores resultados. Para ello, lo que se hace es introducir datos de entrenamiento en la red y en función del resultado que se obtenga se van ajustando los parámetros de acuerdo al error obtenido y en función de cuánto haya contribuido cada neurona a dicho resultado. Este método es conocido como *backpropagation* (propagación hacia atrás) y gracias a él se consigue que la red neuronal aprenda.

Podemos resumir el proceso de entrenamiento de la siguiente manera:

1.Se eligen valores aleatorios para los parámetros de la red (pesos y bias)

2. Para cada pase completo por la red, ida y vuelta, del conjunto de datos de entrenamiento (lo que se conoce como epoch):

   a) Se realiza el forward propagation (propagación hacia adelante) de los datos (la entrada se propaga capa por capa a través de la red hasta calcular la salida final de la red)

   b) Se calcula el coste o pérdida (la diferencia entre el resultado obtenido y el resultado esperado)

c) Se realiza la propagación hacia atrás, para propagar el coste o perdida a cada parámetro de la red

d) Se actualizan los parámetros de la red utilizando la técnica Gradient Descent para lograr que el coste se vaya minimizando en los sucesivos entrenamientos

e)Se vuelve al paso2 *a* hasta que se alcance el número de epoch establecido

Ahora vamos a proceder a explicar con más detalle el proceso de entrenamiento: Para dar inicio a este proceso, los pesos iniciales de la red son elegidos al azar.
Luego se realiza la propagación hacia adelante de los datos de entrenamiento, es decir, la red se expone a la totalidad de los datos de entrenamiento, los cuales cruzan toda la red neuronal para obtener el cálculo de sus predicciones. En este proceso, las neuronas se van activando, aplicando sus transformaciones a los datos recibidos desde las capas anteriores, las cuales van a ser enviadas a las capas siguientes.
Cuando los datos hayan cruzado todas las capas y todas sus neuronas hayan realizado sus cálculos, se alcanzará la capa final con un resultado de predicción para esos datos de entrada.

A continuación, se utiliza una función de coste, previamente elegida en base al tipo de problema, para estimar la pérdida (error) y comparar/medir qué tan bueno o malo fue el resultado de predicción en relación con el resultado correcto. Idealmente, se busca que el error sea cero, es decir, sin diferencia entre el valor estimado y el esperado, lo cual es prácticamente imposible de conseguir, ya que estaríamos hablando de un modelo perfecto. Por lo tanto, a medida que se entrena el modelo, los pesos de las interconexiones de las neuronas se van a ir ajustando gradualmente hasta obtener el menor error posible, es decir, el más cercano a cero.

La funciones de coste generalmente utilizadas para cada tipo de problema son las siguientes [25]:

Tipo de problema	**Tipo de salida**	**Función de activación final**	**Función de coste**
Regresión	Valor numérico	Linear	Mean Squared Error (MSE)
Clasificación	Resultado binario	Sigmoid	Binary Cross Entropy
Clasificación	Una etiqueta,múltiples clases	Softmax	Cross Entropy
Clasificación	Múltiples etiquetas,múltiples clases	Sigmoid	Binary Cross Entropy

Una vez que se ha calculado la pérdida, la misma necesita ser propagada hacia atrás en la red. A partir de la capa de salida, esa información de pérdida se propaga a todas las neuronas de la capa oculta anterior. Sin embargo, las neuronas de esta capa oculta sólo reciben una fracción del valor total de la pérdida, en función de la contribución relativa que cada neurona ha tenido en el resultado final. Este proceso se repite, capa por capa, hasta que todas las neuronas de la red hayan recibido esta información que describe su contribución relativa a la pérdida total.

Luego de difundir esa información por toda la red se procede a ajustar el peso de las conexiones entre las neuronas. Como comentamos anteriormente, se busca que la pérdida sea lo más cercana posible a cero la próxima vez que la red realice las predicciones, es decir, se busca minimizar el valor de la función de coste. Esto se podría realizar mediante el cambio aleatorio de los parámetros de la red hasta que la función de coste devuelva un resultado bajo, lo cual no resulta muy eficiente. En su lugar, se utiliza la técnica denominada *Gradient Descent*.

El *Gradient Descent* es una técnica que permite encontrar el mínimo de una función. En nuestro caso, se busca el mínimo de la función de coste. Funciona cambiando los pesos en pequeños incrementos después de cada pase del conjunto de datos a través de la red. Al calcular el derivado (o gradiente) de la función de coste en un determinado conjunto de pesos, somos capaces de ver en qué dirección se encuentra el mínimo. Gráficamente se puede ver de manera sencilla en la siguiente figura:

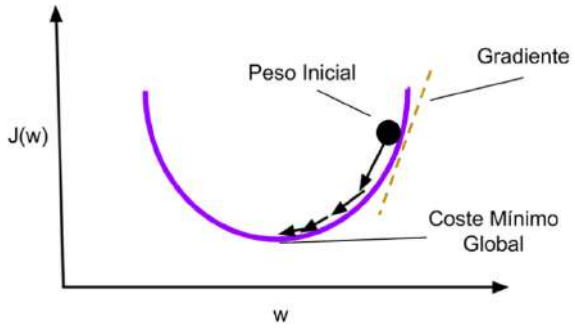


Figura 2.12: Algoritmo de Gradient Descent

Para entender la Figura2.12, imaginemos la siguiente situación: una persona (peso inicial) que realiza trekking desea descender de una montaña (función de coste) hacia un valle (coste mínimo global), y cada paso de la persona está determinado por la inclinación de la pendiente (gradiente) y por la longitud de su pierna (tasa de aprendizaje).

Durante el entrenamiento, la retropropagación del error estima la cantidad de error de la que son responsables los pesos de cada neurona. En lugar de actualizar el peso de esa conexión con la cantidad total calculada, se utiliza la tasa de aprendizaje para aplicar un valor menor.

La tasa de aprendizaje (*learning rate*) es un hiper parámetro que tiene un valor generalmente entre 0.0 y 1.0, y que indica cuánto se actualizan los pesos durante el entrenamiento. En el caso que su valor sea 0.1 por ejemplo, los pesos de la red van a ser actualizados con el 10 % del error de peso estimado. Este parámetro controla la velocidad con la que el modelo aprende, ya que controla el porcentaje de error con el que se actualizan los pesos. En general, un valor elevado de este parámetro permite entrenar más rápidamente al modelo, con la desventaja de llegar a un conjunto no tan óptimo de sus pesos, lo que puede llegar a afectar la efectividad en las predicciones del modelo. En cambio, un valor bajo del parámetro permite al modelo llegar a un conjunto de pesos óptimo, pero como consecuencia el tiempo de entrenamiento requerido por la red va a ser mucho mayor.

Si bien no se puede calcular analiticamente el valor óptimo de la tasa de aprendizaje para un modelo y un conjunto de datos en particular, se suele utilizar 0.1 o 0.01 como punto de partida, el cual luego de un proceso de prueba y error vamos a poder modificarlo (o no) según los resultados que obtengamos en las predicciones durante o al final del entrenamiento.

Luego de realizar la actualización de todos los pesos de la red, se repite el proceso (comenzando desde el paso2 *a* del proceso de entrenamiento) hasta que se alcance el número de epoch establecido.

2.4.Redes Neuronales Convolucionales

Una de las arquitecturas comúnmente utilizada en la actualidad para las tareas de Visión Artificial es la Red Neuronal Convolucional o CNN, la cual es un tipo de red neuronal artificial profunda muy eficiente en tareas de reconocimiento que se utiliza principalmente para tareas de procesamiento de imágenes como la detección e identificación de objetos.

Estas redes son similares a las Redes Neuronales Artificiales: se componen de neuronas que tienen pesos y bias (sesgos); cada neurona recibe algunas entradas, realiza un producto escalar y luego aplica una función de activación; y tienen una función de pérdida o coste sobre la última capa, la cual estará totalmente conectada.

Lo que diferencia a estas redes de las ANN es que están diseñadas específicamente para procesar de manera muy eficiente datos en forma de arreglos multidimensionales: 1D para señales y secuencias (incluido el lenguaje natural); 2D para imágenes o espectrogramas de audio; y 3D para video o imágenes volumétricas, lo que permite codificar ciertas propiedades en la arquitectura, permitiendo ganar en eficiencia y reducir la cantidad de parámetros en la red, logrando disminuir el tiempo y los recursos requeridos para su entrenamiento.

En estas redes, una entrada es pasada a través de un conjunto de capas (convolucionales, no lineales, de pooling y completamente conectadas) con el objetivo de obtener un output, el cual podría ser una clase o una probabilidad de las clases que mejor describan a una imagen por ejemplo.

Las CNN trabajan modelando de manera consecutiva (e incremental) pequeñas regiones de información, y combinando esa información a medida que van avanzando por las distintas capas de la red. En imágenes por ejemplo, la primera capa intentará detectar los bordes, colores simples o curvas. Luego, las capas posteriores tratarán de usar esa información para detectar otras características más complejas como las posiciones de los objetos, la iluminación, las escalas, etc. Por último, las capas finales podrán hacer uso de todas las características detectadas anteriormente y mediante una

imagen de entrada arribar a una predicción final, que podría ser detectar los objetos que contiene y cuáles son sus ubicaciones.

Si bien el origen de estas redes data del año 1980, en donde Fukushima propuso el Neocognitron [26, 27], el cual se conoce como la primer red neuronal convolucional, recién en el 2012 se volvieron populares debido a los resultados obtenidos por la red AlexNet [28], desarrollada por Alex Krizhevsky, que logró ganar la competencia ImageNet (un problema de clasificación de imágenes con más de 1.2 millones de imágenes para clasificar en 1000 categorías diferentes) con una tasa de error del 15.3 % (mejorando ampliamente la tasa anterior de 26.2 %) [29].

2.4.1. Conceptos previos

Antes de continuar describiendo este tipo de redes es necesario considerar lo siguiente:

Clasificación de imágenes Como ya hemos comentado, la clasificación de imágenes es la tarea de procesar una imagen y clasificarla con la categoría o clase (gato, perro, etc) que mejor la describa. Para nosotros, esta tarea de reconocimiento es una de las primeras tareas que aprendemos desde el momento que nacemos y es algo que surge de manera natural y sin esfuerzo cuando somos adultos. Sin siquiera pensarlo demasiado, somos capaces de identificar rápidamente y sin problemas el entorno en el que nos encontramos, así como los objetos que nos rodean. Cuando vemos una imagen o simplemente vemos al mundo que nos rodea, la mayoría de las veces somos capaces de caracterizar inmediatamente la escena observada y darle a cada objeto una "etiqueta", todo sin siquiera ser conscientes de ello.

Estas habilidades mecanizadas de reconocer rápidamente patrones, generalizar a partir de conocimientos previos y de adaptarse a diferentes entornos de una imagen son las que buscamos al utilizar Redes Neuronales Convolucionales.

Tensor Las entradas, salidas y todo lo que se encuentra dentro de las redes neuronales se representa mediante tensores. Los tensores son objetos matemáticos que almacenan valores numéricos y que pueden tener distintas dimensiones. Así, por ejemplo, un tensor de 1D es un vector, uno de 2D es una matriz, uno de 3D es un cubo, etc.

Parámetros En un modelo de Machine Learning existen dos tipos de parámetros: los parámetros del modelo y los hiper parámetros. Los parámetros del modelo son variables internas al modelo y sus valores pueden estimarse a partir de los datos. Un ejemplo de este tipo de parámetro son los pesos de la red. Por otra parte, los hiper parámetros son parámetros externos al modelo y cuyo valor no puede estimarse a partir de datos. La tasa de aprendizaje y el número de iteraciones o epoch son ejemplos de hiper parámetros.

Los hiperparametros deben ser ajustados con el objetivo de obtener un modelo con la mayor precisión posible. El valor de este tipo de parámetros no puede conocerse a priori, por lo que suelen utilizarse valores genéricos que han funcionado anteriormente en problemas similares o valores que surjan de la prueba y error.

Imágenes como inputs de la red neuronal Como comentamos anteriormente, las redes neuronales se manejan con tensores, por lo que toman como entrada un tensor n-dimensional (por ejemplo una matriz) y retornan como salida un tensor m-dimensional.

Cuando trabajan con imágenes, las redes "ven" matrices tridimensionales ($alto \times ancho \times profundidad$). En el caso de una imagen con escala de grises (grayscale) la profundidad del tensor es 1, por lo que el tensor es una matriz cuyo valores son números enteros del 0 (negro) al 255 (blanco) que describen la intensidad del color gris en cada pixel de la imagen. En la Figura 12b se puede observar la intensidad del color gris en cada pixel de la Figura 12a.

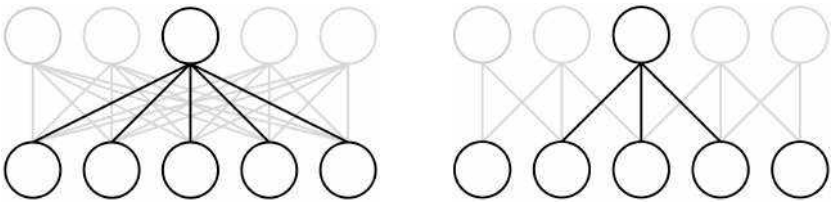
En el caso de las imágenes a color, el tensor posee una profundidad igual a 3, es decir, la imagen es "vista" como una colección de tres matrices bidimensionales, una para cada canal o color: rojo (**R**ed), verde (**G**reen) y azul (**B**lue). Aquí también los

valores de las matrices se encuentran dentro del rango 0-255, los cuales describen la intensidad de cada color en cierto pixel de la imagen.

2.4.2.Arquitectura

Las CNN poseen dos características principales (además de la operación de Convolución y Pooling) que las hacen tan eficientes y exitosas en comparación a las redes neuronales tradicionales:

Conexiones locales En una ANN tradicional cada output de una capa está conectado a todos los inputs de la siguiente capa, en cambio, las CNN poseen interacción reducida (sparse interaction), ya que están basadas en el concepto de conectividad local, por lo que cada output va a conectarse con los inputs más cercanos (o vecinos).



(a) Conexiones en una ANN (b) Conexiones en una CNN

Figura 2.13: Conexiones en ANN y CNN

En la Figura2.13se muestran 5 inputs y 5 nodos en la siguiente capa. En el caso de las ANN tradicionales que están totalmente conectadas, el número de conexiones es 5*5 = 25. En cambio, para la CNN con conectividad local en donde 3 inputs vecinos se conectan a cada nodo de la siguiente capa, el número de conexiones es 3*3 + 2*2 = 13. (los nodos de los extremos en la siguiente capa solo tienen 2 conexiones con los inputs).

Pesos compartidos En una red neuronal tradicional, cada peso se utiliza exactamente una vez cuando se calcula la salida de la capa (se multiplica por un elemento de la entrada y no se vuelve a utilizar). En una CNN, en cambio, cada miembro (peso) del Kernel es utilizado varias veces, para cada ubicación de la entrada, tal como veremos

más adelante.

En cuanto a la estructura de las CNN, estas redes constan de dos partes: una que se encarga de extraer la información de la entrada y otra que se encarga de clasificarla.

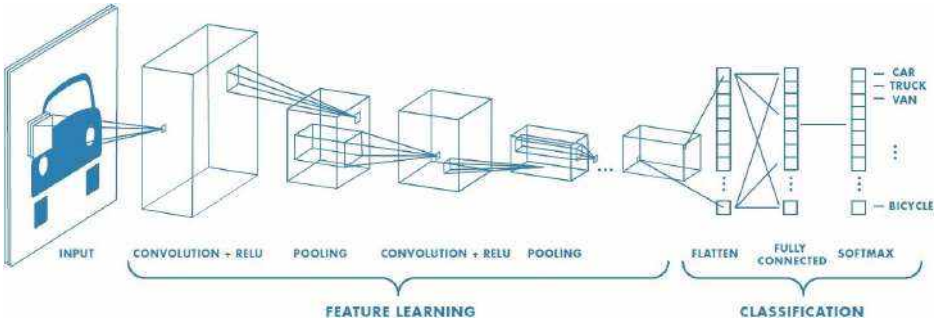


Figura 2.14: Estructura de una CNN [1]

En la primera parte se encuentran las capas de convolución y las de reducción (pooling), las cuales se encargan de "desglosar" la imagen en características (features) analizándolas de manera independiente. La segunda parte contiene una capa totalmente conectada que toma la salida generada por las capas de convolución y reducción y predice la categoría que mejor describa a la imagen.

A continuación vamos a comentar los distintos componentes que posee una CNN:

2.4.3.Capa de Convolución

La primera capa en una CNN siempre es una capa de Convolución, la cual tiene como objetivo extraer características importantes (patrones) de la entrada mediante el uso de filtros (*kernels*), los cuales son las matrices de pesos en una CNN. Las características extraídas son los llamados *feature maps* (mapas de características) y hay que tener en cuenta que diferentes filtros extraen diferentes características de la entrada.

El término convolución se refiere a la combinación matemática de dos funciones para producir una tercera función, y su objetivo es combinar dos conjuntos de información.

En el caso de las CNN, la convolución es realizada sobre los datos de entrada con la ayuda del filtro para producir el mapa de características.

En la convolución se desliza el filtro a través de la entrada y en cada ubicación se realiza una multiplicación de matrices (entre la entrada y el filtro), cuyo resultado se almacena en el mapa de características. Para ilustrar lo mencionado, imaginemos que pasamos una linterna de izquierda a derecha y de arriba hacia abajo, cuya luz cubre un área de 5×5, a través de una imagen a color de $32 \times 32 \times 3$. La linterna es el filtro y la región que está siendo iluminada es el campo receptivo (*receptive field*). El filtro es un arreglo de números (llamados pesos o parámetros), y su profundidad es la misma que la profundidad del input, por lo que en nuestro caso sería $5 \times 5 \times 3$.

Para realizar el proceso de convolución se comienza observando la región de igual tamaño al filtro que se encuentra arriba a la izquierda de la imagen. A medida que el filtro se desplaza (o convoluciona) sobre la imagen, multiplica sus valores con los valores de los píxeles de la imagen. Estas multiplicaciones luego son sumadas para obtener un número que va a ser almacenado en el mapa de características. El mapa de características resultante va a tener un tamaño de $28 \times 28 \times 1$, ya que existen 784 (32×32) posiciones en la que el filtro de 5×5 puede moverse a través de la imagen. La profundidad del mapa es 1 debido a que se aplicó un solo filtro. Más adelante vamos a explicar qué sucede cuando se aplican múltiples filtros a una imagen.

A continuación vamos a mostrar un ejemplo de cómo se realiza la convolución. Supongamos que tenemos una imagen de tamaño 6×6 (en donde cada cuadro representa un píxel de la imagen) y un filtro de 3×3 tal como se muestra en la Figura2.15. Cabe aclarar que para facilitar la explicación se va a mostrar tanto la imagen como el filtro en 2D, pero en realidad las convoluciones se realizan en 3D, tal como vamos a ver luego.

Figura 2.15: Imagen y filtro para convolución

El proceso de convolución comienza observando la región de igual tamaño al filtro que se encuentra arriba a la izquierda en la imagen. Cada cuadro del filtro es multiplicado por cada cuadro de la región observada, siguiendo el orden de las columnas:
La primera columna suma 13 $(3 \times 1 + 5 \times 1 + 5 \times 1)$
La segunda columna suma 0 $(4 \times 0 + 3 \times 0 + 4 \times 0)$
La tercera columna suma 11 $(6 \times 1 + 2 \times 1 + 3 \times 1)$
Por lo tanto, el resultado para la primera posición del mapa de características es 24 $(13 + 0 + 11)$, tal como se muestra en la Figura2.16.

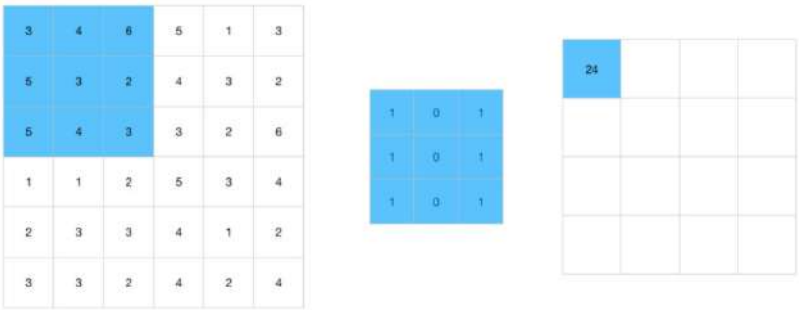


Figura 2.16: Resultado de la primer convolución

Luego se continúa realizando las mismas operaciones hasta llegar a la última columna de la primera fila, tal como se observa en la siguiente figura.

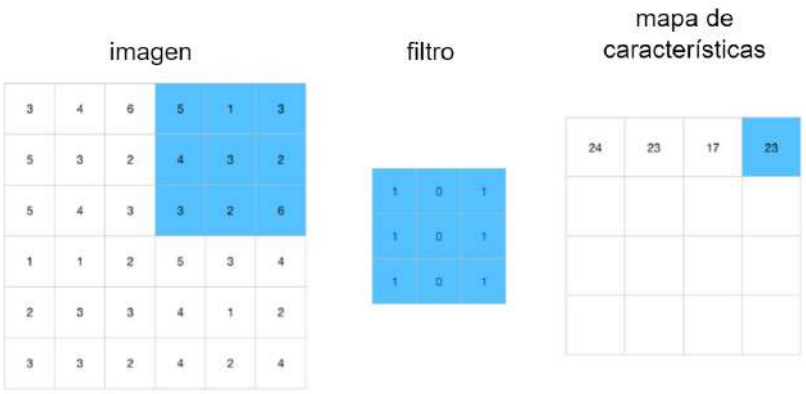


Figura 2.17: Resultado de las convoluciones de la primera fila

Al terminar con las columnas de la primera fila, se debe pasar a la segunda fila de la imagen, y así continuar hasta la última fila y la última columna, obteniendo el siguiente mapa de características.

24	23	17	23
18	20	15	24
16	20	14	24
13	20	13	23

Figura 2.18: Mapa de características resultante

Debemos tener en cuenta que es posible aplicar varios filtros a un mismo input, lo que va a dar como resultado diferentes mapas de características, uno por cada filtro.

Anteriormente comentamos que el mapa de características resultante de aplicar un filtro a una imagen tiene una profundidad de 1. En el caso de aplicar 6 filtros a una imagen se van a generar 6 mapas de características, los cuales van a ser apilados para producir un único tensor de profundidad 6. En la Figura2.19se puede observar una imagen de $32 \times 32 \times 3$ a la cual se le aplican 6 filtros de $5 \times 5 \times 3$. Como resultado

se van a generar 6 mapas de activación de 28×28, los cuales van a ser apilados para formar un tensor de $28 \times 28 \times 6$.

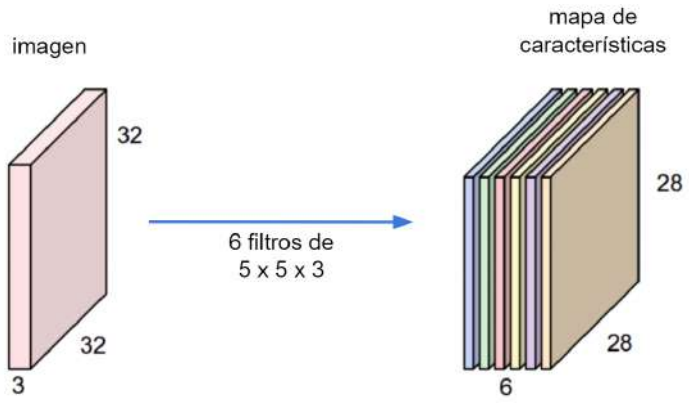


Figura 2.19: Resultado de aplicar 6 filtros a una imagen

Los filtros pueden ser considerados identificadores de características (*feature identifier*). Con características nos referimos a cosas como bordes rectos, colores simples y curvas, características simples que todas las imágenes tienen en común entre sí.

Digamos por ejemplo que tenemos un filtro detector de curvas. Como detector de curvas, el filtro generará valores numéricos más altos para las regiones de la imagen en donde se encuentren curvas. Si en la imagen de entrada hay una forma que se asemeje a la curva que representa este filtro, todas las multiplicaciones sumadas resultarán en un valor alto; en cambio, si en la imagen no hay ninguna forma que se asemeje a una curva, el valor resultante va a ser menor.

Ese es solo un ejemplo de un filtro que detecta curvas. Podríamos tener otros filtros para identificar líneas rectas u otro tipo de formas. Cuantos más filtros, mayor va a ser la profundidad del mapa de características y más información vamos a disponer sobre el input.

Para finalizar, los filtros no solo detectan características de bajo nivel. Como ya sabemos, la entrada de la primera capa de convolución es la imagen original, y la salida es uno o varios mapas de características. Si a esos mapas resultantes se le aplica un nuevo conjunto de filtros (se los pasa a través de una segunda capa de Convolución), la salida serán mapas que representan características de más alto nivel, como por ejemplo

semicírculos (combinación de curvas y bordes rectos), o cuadrados (combinación de varios bordes rectos). A medida que se avanza por la red, y se va atravesando más capas de Convolución, se obtendrán mapas de características que representan características cada vez más complejas. Al final de la red va a haber filtros que, por ejemplo, se activen sólo al detectar personas.

Convoluciones en imágenes a color Tal como hemos mencionado anteriormente, una imagen a color consta de 3 canales, uno para cada color RGB. Esto quiere decir que para una imagen a color, tendremos 3 matrices representándola, una para el color rojo, otra para el verde y otra para el azul. Si tenemos una imagen de 6×6, el tamaño será $6 \times 6 \times 3$. Además, si el filtro es de 3×3, su tamaño será de $3 \times 3 \times 3$, ya que habrá una matriz para cada canal de color (recordemos que la profundidad del filtro debe ser igual a la profundidad de la imagen).

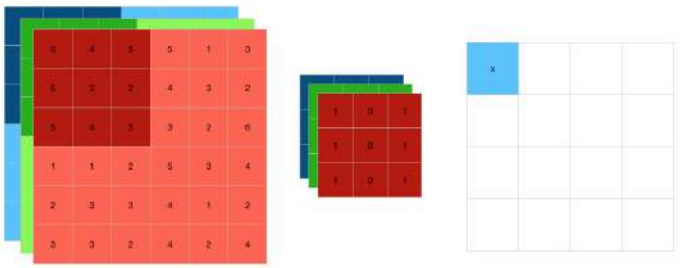


Figura 2.20: Convolución utilizando 3 dimensiones

La convolución en este caso es realizada por canales. La matriz roja del filtro va a operar con el canal rojo de la imagen, la matriz verde lo va a hacer con el canal verde y la matriz azul lo hará con el canal azul. El mapa de características va a tener una profundidad de 1, al igual que cuando se trabaja con filtros de profundidad 1.
Luego de obtener los resultados para cada canal de color, se va a realizar la suma de ellos y el valor resultante va a ser alojado en la posición que corresponda del mapa de características.

Padding El padding es un parámetro importante de la capa de convolución. Su objetivo es agregar píxeles con valor cero alrededor de una imagen, tal como se observa en la figura2.21.

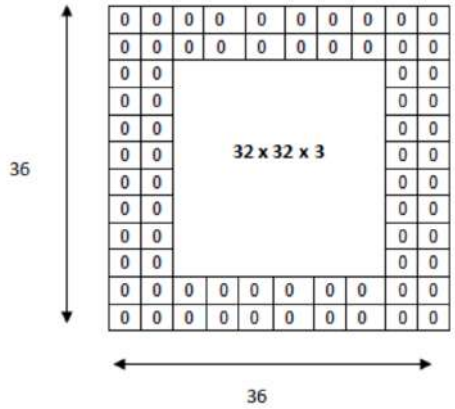


Figura 2.21: Padding en una imagen

La idea detrás del uso del padding es la siguiente. En algunas imágenes, la información importante puede encontrarse en las esquinas o bordes, es decir, lejos del centro. Al realizar la operación de convolución, el filtro pasará muy pocas veces por esas ubicaciones, en comparación a las veces que pasará por el centro de la imagen. Entonces, es necesario agregar píxeles alrededor de la imagen para conseguir "mover" esa información relevante más hacia el centro, o lo que es lo mismo, alejarla de los bordes.

Otro de los usos de este parámetro es controlar el tamaño del mapa de características resultante. Cada vez que se aplica un filtro la matriz generada va decreciendo en dimensión, por lo que luego de aplicar varios filtros, el tamaño del mapa de características va a decrecer rápidamente. Además, hay situaciones en las que deseamos que se preserve en las primeras capas de la red tanta información sobre el input como sea posible, para lograr por ejemplo una mejor extracción de las características de bajo nivel.

Para evitar estos problemas, si tenemos una imagen de $32 \times 32 \times 3$ y queremos que la matriz resultante refleje el mismo tamaño, debemos agregarle un padding de 2 a la imagen original (generando una imagen de $36 \times 36 \times 3$, como se observa en la Figura

2.21). Entonces, al aplicar un filtro de $5 \times 5 \times 3$ con stride 1, el volumen resultante será de $32 \times 32 \times 3$.

Stride Este parámetro indica la cantidad de píxeles en la que se va a desplazar el filtro a través del input. Si el valor del stride es 1, el filtro se va a desplazar de a 1 píxel a la vez; si es 2, se va a desplazar de a 2 píxeles a la vez y así sucesivamente.

2.4.4.Capa ReLU

Las capas ReLU son generalmente combinadas con las capas convolucionales, por lo que podemos decir que trabajan juntas.

Una vez que los mapas de características son generados por la capa de convolución, el siguiente paso es moverlos a una capa ReLU (Figura2.11), la cual les aplica la función ReLU que básicamente convierte todos los píxeles negativos a 0. La salida de esta operación se denomina mapa de características rectificado (*rectified feature map*).

El uso de capas ReLU posee dos ventajas:

- Se introduce la no linealidad en la red. Tanto las convoluciones como la multiplicación de matrices y sumas son lineales, por lo que si no se consigue obtener la no linealidad, el modelo final lineal fallará en la tarea de clasificación.
- Se acelera el proceso de entrenamiento al evitar el problema de desvanecimiento de gradiente (*vanishing gradient*) [30], el cual ocurre en redes con muchas capas, en donde a los pesos que se encuentran más cercanos al input le llega una proporción de error tan pequeña que dificulta que actualicen su valor [31].

2.4.5.Capa de Pooling

Los mapas de características rectificados generados por la capa ReLU son generalmente enviados a la capa de Pooling. La función de esta capa es disminuir el tamaño de la entrada para acortar el tiempo de entrenamiento, mediante la reducción del número

de parámetros y del cómputo requerido por la red, manteniendo siempre la información importante del input.

La operación más común de pooling es el *Max Pooling*, el cual toma el valor máximo de cada región de la imagen. Existe además otra operación denominada *Average Pooling*, que calcula el promedio de los valores de la región.

En la siguiente figura se puede observar una imagen de 4×4 y el resultado de las dos operaciones de Pooling, en donde se hace uso de un filtro de 2×2 con stride 2.

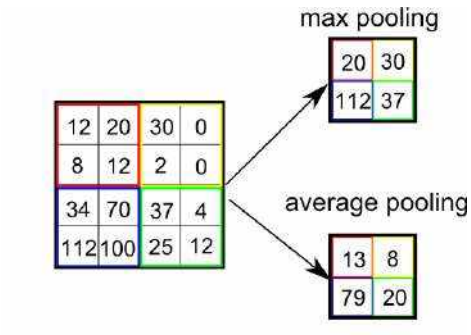


Figura 2.22: Operaciones de Pooling

En la figura se observa que el filtro va a pasar por cuatro regiones de la imagen. Si tomamos a modo de ejemplo la primer región, en el caso de *Max Pooling* el valor resultante será de 20, el cual es el más alto. En el caso de *Average Pooling*, el valor será 13, ya que es el promedio de todos los valores de la región ((12 + 20 + 8 + 12) / 4 = 13).

2.4.6.Capa totalmente conectada

Las capas totalmente conectadas son utilizadas para hacer la clasificación final en la CNN y trabajan de la misma manera que lo hacen en una red neuronal común.

Antes de enviar los datos a la primera capa totalmente conectada de la red se los debe “aplanar” para convertirlos en un vector de 1 dimensión, tal como se muestra en la Figura2.23, ya que esta capa así lo requiere.

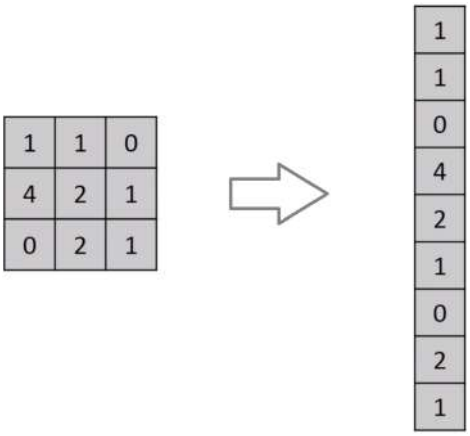


Figura 2.23: Conversión de matriz a vector

Es importante aclarar que una CNN puede poseer varias capas totalmente conectadas al final de la red, y que la última de ellas va a hacer uso de una función de activación, como *softmax* o *sigmoid*, para devolver las predicciones en forma de probabilidades.

2.4.7.Entrenamiento de la red

El proceso de entrenamiento de una CNN es similar al de las redes neuronales tradicionales. Si bien existe cierta complejidad adicional debido al uso de nuevas capas, el entrenamiento es muy parecido: en cada iteración se realiza una propagación hacia adelante de los datos a través de la red, se obtiene un resultado, se lo compara con el resultado esperado y se transmite el error hacia atrás mediante el uso de la propagación hacia atrás. En el caso de las ANN se actualizan los pesos de las conexiones de las neuronas; en cambio, en las CNN se actualizan los pesos de los filtros y de las capas totalmente conectadas que se encuentran al final de la red.

División de los datos El conjunto de datos a ser utilizado por la red neuronal suele dividirse en tres subconjuntos: uno para el entrenamiento, otro para la validación y otro para la prueba. El conjunto de entrenamiento es utilizado por la red para aprender. El conjunto de validación es empleado para realizar predicciones de manera periódica con el objetivo de ir ajustando los diferentes parámetros de la red de acuerdo a los

resultados obtenidos. Por último, el conjunto de prueba es utilizado al final del proceso de entrenamiento con el fin de evaluar el rendimiento real del modelo.

La separación entre los conjuntos de validación y de prueba es necesaria ya que el proceso de entrenamiento está basado en la performance del conjunto de validación. Como vamos a ver a continuación, puede ocurrir que la red "memorice" los datos de ese conjunto, por lo que para realizar una evaluación real del modelo es necesario realizar las predicciones sobre un conjunto de datos totalmente nuevo, que no haya sido visto con anterioridad.

Overfitting El objetivo de los algoritmos de machine learning es el de obtener patrones a partir de los datos de entrenamiento, lo cual le va a permitir predecir o inferir datos nuevos de manera correcta.

El *overfitting* ocurre cuando un modelo "memoriza" o aprende demasiado bien los datos utilizados durante el entrenamiento, generando que tenga dificultades a la hora de reconocer nuevos datos.

Si disponemos de un modelo que presenta un 99 % de precisión sobre su conjunto de datos de validación, pero solo un 30 % sobre su conjunto de prueba, es una clara señal que el modelo fue incapaz de identificar patrones durante el entrenamiento. El *overfitting* puede ser causado como consecuencia de un sobre-entrenamiento del modelo o por la existencia de muestras que resulten no ser del todo representativas. Para intentar reducir este problema existen algunas estrategias, tal como la ampliación del conjunto de datos y la inserción de métodos de regularización.

Un modelo entrenado con una gran cantidad de datos generalmente va a ser capaz de generalizar mejor, por lo que la primer estrategia consiste en ampliar el conjunto de datos. En el caso que no sea posible la obtención de nuevos datos, se puede hacer uso de la técnica de aumento, la cual consiste en generar nuevas imágenes a partir de las ya existentes a través de la aplicación de diferentes transformaciones (zoom, ruido, cambio de brillo y color, etc).

La segunda estrategia para reducir el *overfitting* es la incorporación de métodos de regularización. Uno de ellos es el *dropout*, el cual consiste en "ignorar" una cierta cantidad de neuronas de manera aleatoria durante el proceso de entrenamiento.

El valor del *dropout* varía entre 0 y 1. Un valor de 0.3 significa que el 30 % de las neuronas de la red no van a intervenir durante el entrenamiento. Tal como sabemos, las neuronas tienen un valor de salida que envían a las neuronas de las siguiente capas. Si al valor de salida se le aplica un *dropout* de 0.3 significa que tiene un 30 % de probabilidad de convertirse en 0, produciendo la "desactivación" de la neurona. El objetivo de ignorar ciertas neuronas de manera aleatoria es otorgarle al modelo una mayor dificultad al intentar memorizar los datos.

Transfer Learning El Transfer Learning es una técnica muy utilizada en machine learning que consiste en utilizar un modelo ya entrenado (generalmente con miles o millones de datos) como punto de partida para resolver nuestro problema.

Esta técnica nos brinda varias ventajas, entre las que destacamos:

- Se necesita menor cantidad de datos de entrenamiento: comenzar a entrenar un modelo desde cero requiere una enorme cantidad de datos y trabajo. Si se desea crear un modelo capaz de detectar bigotes en personas, el modelo primero necesitará aprender a detectar caras y solo entonces podrá aprender a detectar sus características, como los bigotes. En cambio, si usamos un modelo que ya aprendió a detectar caras y lo entrenamos nuevamente para detectar bigotes, podemos lograr el mismo resultado usando solamente imágenes de personas que poseen bigotes.

- El tiempo y los recursos para entrenar el modelo disminuyen drásticamente: Este ítem se encuentra vinculado al anterior. Al contar con un modelo ya entrenado se ahorra una gran cantidad de tiempo y recursos (GPU por ejemplo) en comparación a si quisiéramos entrenar un modelo desde cero. Mediante el uso de transfer

learning es posible entrenar un modelo que detecte nuevos objetos con una gran precisión luego de unas pocas horas de entrenamiento.

- Logra que el machine learning sea más accesible: trabajar con transfer learning facilita el uso del machine learning ya que no es necesario ser un experto en ese campo para poder generar modelos que cuenten con una gran precisión en la detección de objetos, por ejemplo.

La selección del modelo entrenado a utilizar dependerá del problema a resolver y de varios factores, tal como su velocidad, su precisión y si está optimizado para dispositivos móviles. Estos modelos suelen utilizar arquitecturas conocidas, como MobileNet [32], ResNet [33], SSD [34], Inception [35] y Faster RCNN [36]. Además, suelen estar entrenados en enormes datasets, como COCO [1], Open Images[2] e ImageNet[3] por citar algunos ejemplos.

Existen diferentes estrategias para la utilización del transfer learning. Como vimos al comienzo del capítulo, una red convolucional consta de dos partes: una de extracción de características y otra de clasificación. Una estrategia de transfer learning consiste en "congelar" las capas de la red que se encargan de realizar la extracción de características, para así aprovechar su conocimiento y evitar que sus pesos se modifiquen durante el nuevo entrenamiento. En cuanto a la parte de clasificación de la red ya entrenada, es necesario reemplazar su función de activación, ya que nos interesa que la red detecte solamente nuestros nuevos objetos. El entrenamiento entonces se va a realizar sobre las capas totalmente conectadas (según la estrategia elegida), sirviéndose del conocimiento de los pesos de las capas congeladas y retornando resultados a través de la nueva función de activación.

[1] http://cocodataset.org
[2] https://storage.googleapis.com/openimages/web/index.html
[3] http://image-net.org

3. Solución propuesta

El presente trabajo propone el desarrollo de una plataforma cloud de Machine Learning que posibilite la detección de objetos en tiempo real a través de la cámara de un teléfono celular. La plataforma abarca el proceso entero de entrenamiento y uso de un modelo de Machine Learning y posee las siguientes características:

- Todos sus componentes son de uso gratuito. Cualquier persona que desee utilizarla va a poder hacerlo sin la necesidad de invertir dinero.
- Cada componente tiene su función, entradas y salidas bien definidas, lo que permite realizar el intercambio de alguno de ellos de manera sencilla sin la necesidad de realizar grandes cambios en la plataforma.
- Es configurable y extensible. Se proveen numerosos parámetros en las diferentes etapas, permitiendo que el usuario pueda configurarla como lo desee. Además, es posible añadir funcionalidad adicional entre los distintas actividades de manera sencilla.
- Puede ser aplicada a cualquier dominio de interés ya que es genérica en cuanto a lo que puede detectar. El objetivo de la plataforma es detectar objetos y si bien en este trabajo la utilizamos para la detección de síntomas de enfermedades y plagas en cultivos, es posible entrenarla con cualquier conjunto de datos etiquetados sin ningún inconveniente y sin la necesidad de realizar cambios.
- El uso de internet por parte de la aplicación móvil es requerido solo para la descarga del modelo de Machine Learning generado por la plataforma. Una vez descargado el modelo, la aplicación no requiere conectarse a ninguna red de datos

para funcionar, lo que le permite ser ejecutada en zonas sin cobertura móvil por ejemplo.

- Es posible resumir entrenamientos que hayan sido interrumpidos. Este ítem es algo fundamental cuando se trabaja con el entrenamiento de modelos de Machine Learning. Considerando que los entrenamientos pueden tomar horas, o incluso días, la plataforma provee la opción de poder resumir procesos de entrenamiento que hayan sido interrumpidos por el usuario o por el sistema. Esto se logra mediante el guardado de checkpoints en Google Drive y mediante una opción (en el documento de Colab encargado del entrenamiento) que brinda la posibilidad de resumir o empezar un entrenamiento.

- La aplicación móvil busca ser lo más intuitiva posible. Esto responde a la necesidad de poder ser accesible para la mayor cantidad de personas, por ende se busca que la aplicación sea intuitiva al usuario, contando con opciones y mensajes claros y proveyendo feedback ante cada acción del usuario y del sistema.

- Para el entrenamiento del modelo se hace uso de la técnica de Transfer Learning. Si bien la plataforma propone algunos modelos ya entrenados, el usuario puede seleccionar cualquier modelo del repositorio de TensorFlow [1]. Allí figuran más de 40 modelos listos para utilizar, separados por datasets y en donde se especifica para cada uno las arquitecturas de red neuronal que utiliza y la velocidad y precisión en la predicción.

3.1.Diseño

Para lograr que la plataforma abarque el proceso completo de entrenamiento y uso del modelo de Machine Learning se han contemplado las siguientes actividades en la misma:

[1]https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md

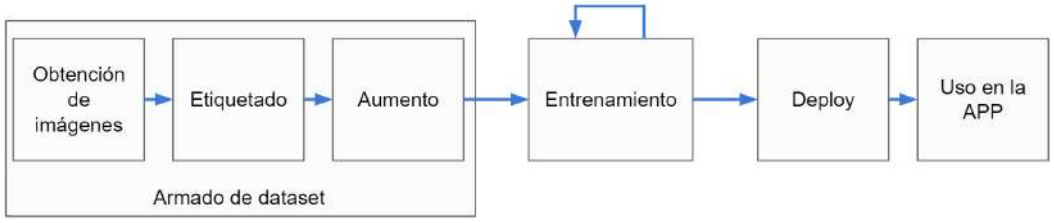


Figura 3.1: Actividades contempladas en la plataforma

En la Figura3.1se observan las 4 actividades principales: el armado del dataset de imágenes, el entrenamiento de la red neuronal, el deploy del modelo generado y el uso del mismo por parte de la aplicación móvil.
El armado del dataset incluye la obtención, el etiquetado y el aumento de las imágenes a ser utilizadas para el entrenamiento.
El entrenamiento es un proceso iterativo durante el cual el algoritmo provee información y diferentes métricas al usuario referidas al desempeño del modelo que se está entrenando. En base a esa información el usuario puede optar por realizar ajustes en la red, en los parámetros, en los datos de entrada, o incluso puede optar por utilizar un modelo diferente, en el caso que los resultados devueltos no sean satisfactorios.
El deploy del modelo consiste en dos actividades. La primera es la generación del modelo a partir de los archivos generados durante el entrenamiento. La segunda actividad es el almacenamiento del modelo en un servicio de almacenamiento cloud.
Por último, el uso del modelo por parte de la aplicación contempla la sincronización (descarga y actualización) del modelo en el teléfono celular y su utilización.

Almacenar un modelo de machine learning en la nube presenta algunas ventajas: el tamaño de la aplicación móvil va a ser menor ya que no incluye ningún archivo (modelo) en su código fuente y, además, en el caso de querer actualizar el modelo no es necesario generar una nueva versión de la aplicación con el modelo actualizado, ya que el mismo puede ser descargado desde internet.

Para cumplir con las actividades mencionadas la plataforma posee los siguientes componentes y sus respectivas conexiones:

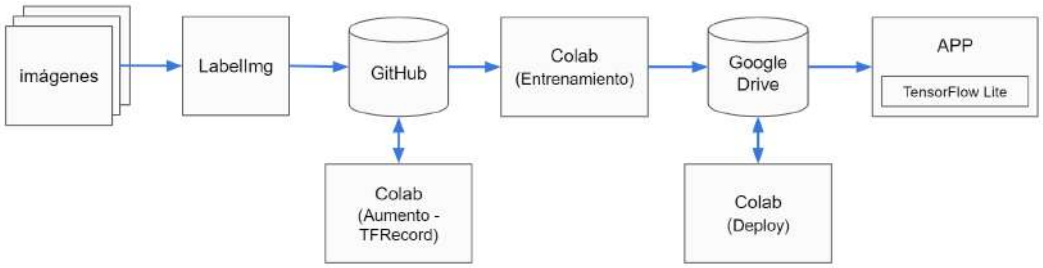


Figura 3.2: Componentes de la plataforma

El flujo de trabajo y de datos en la arquitectura es el siguiente:

1.Primero se debe recolectar las imágenes a ser utilizadas para el entrenamiento.

2. Luego se deben etiquetar las imágenes utilizando el software *LabelImg*, el cual generará un archivo de mapeo .xml para cada una de ellas.

3. Una vez etiquetadas las imágenes, se las debe dividir en 2 conjuntos (entrenamiento y testing). Ambos conjuntos deben ser subidos a un repositorio de Github.

4. Un documento de Colab se va a encargar de realizar el aumento de las imágenes subidas a Github. Luego de ello, generará los archivos necesarios para efectuar el entrenamiento de la red, los cuales subirá a Github.

5. Un segundo documento de Colab, haciendo uso de los archivos generados en el punto anterior, va a realizar el entrenamiento de la red. Durante el entrenamiento se irán generando *checkpoints*, los cuales serán subidos a Google Drive.

6. Un tercer documento de Colab va a obtener el *checkpoint* más reciente desde Google Drive y va a generar el modelo TFLite, el cual va a dejar disponible para su descarga en ese mismo servicio de almacenamiento.

7. La aplicación móvil va a realizar la descarga del modelo desde Google Drive. Una vez descargado, la aplicación va a estar lista para comenzar a detectar objetos en tiempo real a través de la cámara del teléfono.

A continuación describiremos paso a paso las distintas actividades de la plataforma, desde el armado del dataset de entrenamiento hasta el uso del modelo en la aplicación. En cada etapa explicaremos los componentes involucrados y el intercambio de información dentro de la misma.

3.1.1. Armado del dataset de entrenamiento

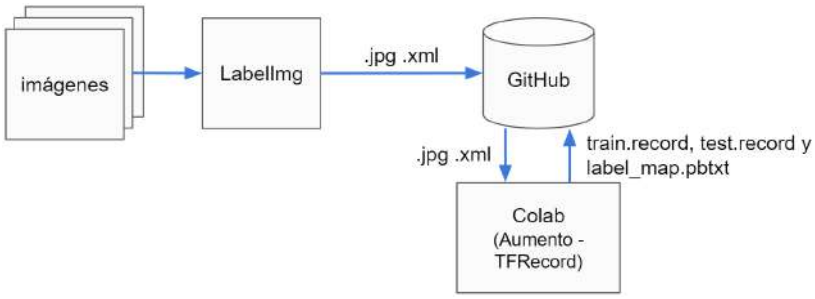


Figura 3.3: Armado del dataset de entrenamiento

Output: archivos *train.record*, *test.record* y *label_map.pbtxt* subidos al repositorio de GitHub.

La primera actividad de la plataforma contempla la obtención de las imágenes, su etiquetado, su aumento y la generación de los archivos requeridos por la actividad de entrenamiento.

Obtención de imágenes Lo primero que debemos hacer es obtener las imágenes que van a ser utilizadas para el entrenamiento de la red. En lo posible deben ser imágenes con buena definición y en donde los objetos de interés se visualicen de forma clara. Es deseable no incluir imágenes que posean zoom, ruido, poca luminosidad, excesivo brillo, etc. ya que el aumento de imágenes se va a encargar de generar esas situaciones.

En lo posible debe agregarse, además, imágenes que no posean los objetos que se desean detectar. El objetivo de agregar este tipo de imágenes es el de reducir la cantidad de falsos positivos, situación en la que el modelo detecta erróneamente los objetos. La variedad de imágenes a incluir dependerá de varios factores, tal como el tipo de objeto a detectar, y lo ideal es incluir tanto imágenes relacionadas al contexto de los objetos como imágenes no relacionadas en lo absoluto al mismo.

Antes de proceder a etiquetar las imágenes es conveniente redimensionarlas a un tamaño único, generalmente al tamaño de la imagen mas chica, para que ocupen menos espacio en el almacenamiento. La redimensión va a permitir, además, reducir la cantidad de parámetros y cómputo utilizado durante el entrenamiento de la red neuronal. Para realizar el redimensionado de las imágenes existen muchas herramientas gratuitas. En Windows podemos utilizar el software *Image Resizer for Windows*[2] que permite seleccionar la totalidad de las imágenes y con pocos clicks redimensionarlas al tamaño elegido.

Etiquetado de imágenes El tipo de aprendizaje requerido por la red convolucional es supervisado, y tal como comentamos en la sección de Machine Learning, este aprendizaje utiliza datos etiquetados.

Cuando se trabaja con imágenes y detección de objetos una de las maneras que tenemos de proveer datos etiquetados es a través de archivos de mapeo .xml. Entonces, la imagen servirá como dato de entrada, y el archivo .xml como resultado.

Como se observa en el fragmento de código3.1, en el archivo de mapeo .xml se detallan los datos de la imagen (nombre, tamaño) y se listan los objetos etiquetados que contiene, especificando sus nombres y las coordenadas de sus cuadros delimitadores (*bounding boxes*).

[2]https://www.bricelam.net/ImageResizer

```
<annotation>
  <folder>training</folder>
  <filename>imagen_con_cladisporium.jpg</filename>
  <size>
    <width>640</width>
    <height>480</height>
    <depth>3</depth>
  </size>
  <object>
    <name>cladisporium</name>
    <bndbox>
      <xmin>225</xmin>
      <ymin>206</ymin>
      <xmax>371</xmax>
      <ymax>283</ymax>
    </bndbox>
  </object>
</annotation>
```

Fragmento de código 3.1: Estructura del archivo de mapeo Pascal VOC

Para la generación de estos archivos se va a utilizar el software *LabelImg* [3], un anotador de imágenes gratuito, ligero y sencillo de utilizar que se encuentra disponible para Windows, Mac y Linux.
Cabe aclarar que este software es el único componente de la plataforma que no es cloud, debido a que las herramientas Web disponibles al momento de realizar este trabajo presentan errores al manejar gran cantidad de imágenes o son pagas, lo que no cumple con nuestro requerimiento de utilizar únicamente servicios gratuitos.

El uso de esta herramienta es fácil e intuitiva. En la figura3.4se puede observar su pantalla principal, en la cual aparecen 2 objetos etiquetados (con la clase “oidio”).

Para realizar el etiquetado de todo el conjunto de imágenes se deben seguir los siguientes pasos:

1. Verificar que se encuentre seleccionada la opción “*PascalVOC*” en el menú de la izquierda.

2. Hacer click en “*Open Dir*” y seleccionar la carpeta que contiene las imágenes. Al hacerlo se van a listar todas las imágenes en “*File List*”.

[3]https://github.com/tzutalin/labelImg

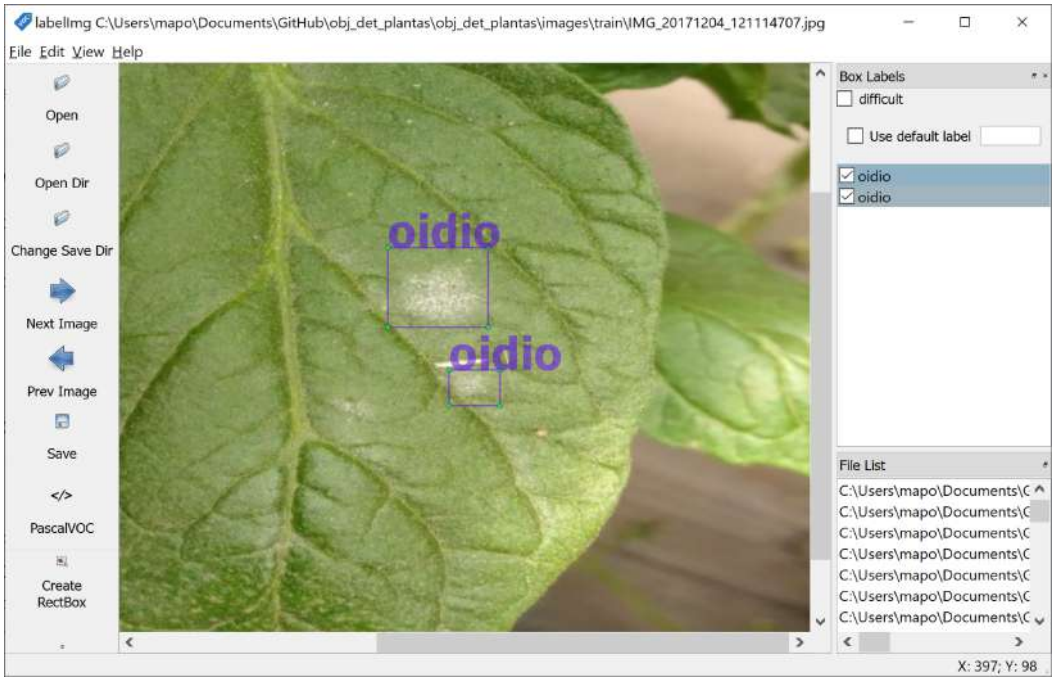


Figura 3.4: Pantalla principal de LabelImg

3. Hacer click en "*Change Save Dir*" y escoger la misma carpeta del punto anterior, para así generar los archivos de mapeo en el mismo directorio donde se encuentran las imágenes.

4.Empezar a etiquetar las imágenes.

5. Para etiquetar los objetos basta con seleccionar la opción "*Create RectBox*" y marcar el objeto en la imagen arrastrando el mouse. Una vez marcado el objeto se debe ingresar el nombre del mismo. Hay que prestar atención al introducir los nombres, ya que si se comete un error en el tipeo del mismo el entrenamiento de la red se verá afectado.

   En el caso que la imagen no posea objetos se debe seleccionar la opción "*Verify Image*" (que se encuentra en la barra de herramientas) para indicar que se genere el archivo de mapeo con el listado de objetos vacío.

6.Presionar " *Save*" para generar el archivo de mapeo.

7. Presionar "*Next Image*". Repetir el proceso desde el paso5hasta que se hayan etiquetado todas las imágenes.

Si bien parece un proceso engorroso en el caso que contemos con una gran cantidad de imágenes a etiquetar, la herramienta provee varios atajos de teclado, lo cual permite ahorrar tiempo en el proceso del etiquetado.

Luego de etiquetar las imágenes se las debe dividir en 2 carpetas (training y test). La primera se va a utilizar para el entrenamiento, y la segunda para ir evaluando la performance del entrenamiento de manera periódica, lo cual le va a permitir al algoritmo realizar los ajustes necesarios en la red con el fin de mejorar las predicciones. El porcentaje de distribución entre ambas carpetas suele ser de 80 % para training y 20 % para test, sin embargo el porcentaje queda a elección del programador, de acuerdo a sus preferencias o necesidades.

Luego de dividir los archivos se los debe almacenar en la nube, para ello se va a hacer uso de Github [4]. Este servicio provee repositorios gratuitos (tanto públicos como privados) con un limite de 100MB por archivo subido y de 1GB como máximo (recomendado) por repositorio [5], lo cual es más que suficiente para nuestro uso.

Aumento de imágenes Lo que debemos hacer a continuación es ampliar el conjunto de imágenes etiquetadas. Para ello, vamos a utilizar la técnica de aumento, que nos permite generar nuevas imágenes a partir de las ya existentes. Esta técnica es realmente útil cuando disponemos de pocas imágenes o cuando son difíciles de etiquetar. Su importancia radica en que permite incrementar la eficacia del modelo, ya que se va a proveer una gran cantidad de imágenes extras que contemplan diferentes condiciones o situaciones que seguramente no estaban contempladas en las imágenes originales. También es útil para evitar el *overfitting*, es decir la memorización por parte del modelo de las imágenes de entrenamiento, lo que lleva a que la precisión en la detección de nuevas imágenes sea baja.

[4]https://www.github.com
[5]https://help.github.com/es/github/managing-large-files/what-is-my-disk-quota

El aumento genera imágenes realizando transformaciones a la imagen original. Es posible por ejemplo rotar, agregar zoom, cambiar los colores o la iluminación, agregar ruido, etc.

La modificación de una imagen conlleva a que tengamos que actualizar también su archivo de mapeo, ya que si por ejemplo volteamos y rotamos una imagen, las posiciones de los objetos que contiene van a cambiar.

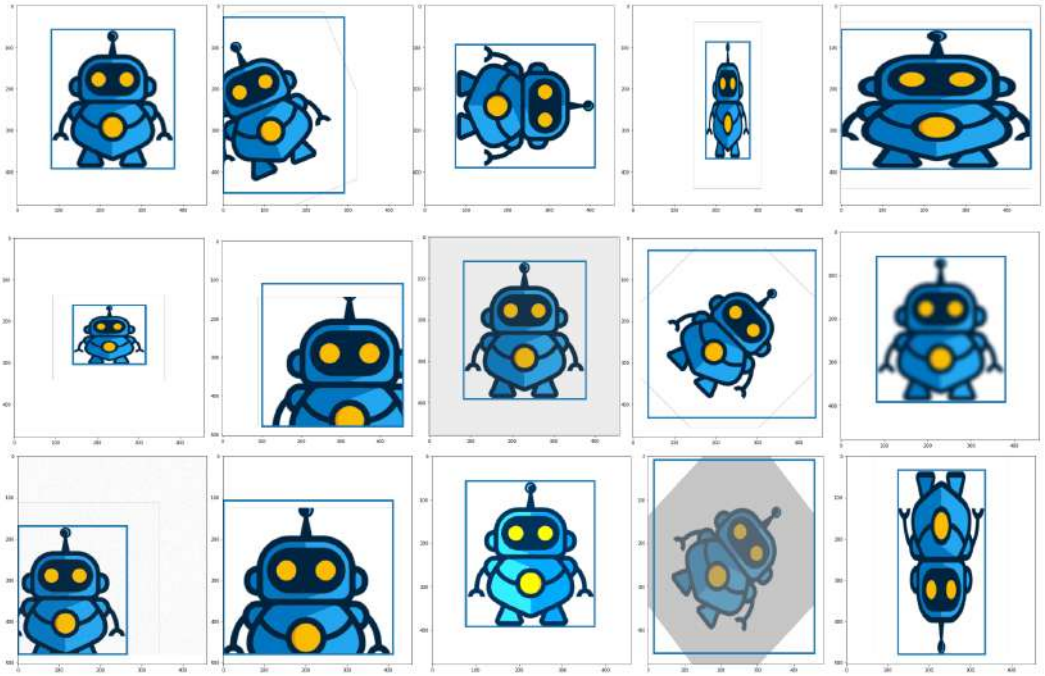


Figura 3.5: Ejemplo de aumento de imagen

En la Figura3.5se puede observar que la primer imagen es la original y las demás imágenes son las generadas automáticamente a través de transformaciones. En ellas, los cuadros delimitadores fueron actualizados para seguir marcando de manera correcta al objeto.

Este proceso de aumento lo vamos a realizar utilizando el servicio Google Colaboratory [6] (Colab), el cual es un entorno gratuito de Jupyter Notebook [7].

Jupyter Notebook es una aplicación web de código libre que permite crear y compartir documentos que poseen código. Cada documento está compuesto por celdas,

[6]https://colab.research.google.com
[7]https://jupyter.org

las cuales pueden contener código o texto y su resultado (texto, gráficos, tablas, resultados de operaciones) se muestra luego de cada una de ellas. Al soportar Python es ampliamente utilizado en tareas de Machine Learning.

Google Colab, por su parte, no requiere configuración, se ejecuta completamente en la nube y provee en su versión gratuita una GPU K80, 12GB de memoria RAM y 12 horas de uso continuo hasta que se reinicie el entorno. Además, provee un filesystem, cuenta con librerías pre instaladas listas para ser utilizadas y permite conectarse a servicios como Github y Google Drive de manera sencilla.

Ejecutar documentos de Colab en la nube presenta varias ventajas. Una de ellas es que se hace uso del hardware de Google, lo cual garantiza una alta estabilidad y disponibilidad de los recursos. Otra ventaja es que permite que una persona que posea hardware no adecuado para entrenar modelos de Machine Learning, pueda hacerlo, ya que otorga un gran poder de cómputo de manera gratuita.

El documento de Colab que se encarga de generar el aumento de las imágenes se encuentra en el anexo de códigos fuentes, bajo el nombre de "AUMENTO_Y_ TFRECORDS.ipynb". Además, enhttp://mapo-lp.github.io/aumento.htmlse encuentra una copia del mismo, lista para ser ejecutada en Google Colaboratory. El funcionamiento de este documento es el siguiente:
Primero se deben especificar ciertos parámetros, tal como la cantidad de imágenes a generar y los datos del repositorio de Github donde se encuentran las imágenes y donde se van a subir los archivos generados por esta actividad.
Para la generación de las nuevas imágenes es necesario aplicar transformaciones a las imágenes originales. Las transformaciones son realizadas por la librería albumentations, la cual posee una gran cantidad de posibles transformaciones, desde agregar zoom y cambiarle el color a una imagen, hasta agregarle efecto de lluvia, neblina o nieve.
El documento de Colab posee un listado de transformaciones por defecto (Fragmento de código3.2), del cual se van a tomar 2 de ellas al azar para la generación de cada imagen.

```
aug = iaa.SomeOf(2, [
    iaa.Affine(scale=(0.5, 1.5)),
    iaa.Affine(rotate=(-60, 60)),
    iaa.Affine(translate_percent={"x":(-0.3, 0.3),"y":(-0.3, 0.3)}),
    iaa.Fliplr(1),
    iaa.Multiply((0.5, 1.5)),
    iaa.GaussianBlur(sigma=(1.0, 3.0)),
    iaa.AdditiveGaussianNoise(scale=(0.03*255, 0.05*255))
])
```

Fragmento de código 3.2: Transformaciones a realizar en imágenes

Al generar cada nueva imagen se va a crear también su archivo de mapeo con las posiciones de los objetos actualizadas.
Es importante aclarar que el aumento será realizado sobre las imágenes que posean objetos etiquetados, por lo que las imágenes que no contienen los objetos de interés no serán aumentadas.

Una vez creadas todas las imágenes se van a generar los archivos *train.record*, *test.record* y *label_map.pbtxt*, los cuales son requeridos por la red neuronal para realizar el entrenamiento. Los dos primeros archivos poseen el formato *TFRecord*, el cual es un formato de archivo binario orientado a registros, que permite el almacenamiento y el procesamiento eficiente de grandes conjuntos de datos. El archivo restante es un archivo de texto en donde se especifica el ID y el nombre de los objetos etiquetados. Su estructura se muestra a continuación:

```
item {
    id: 1
    name: 'objeto1'
}
item {
    id: 2
    name: 'objeto2'
}
```

Fragmento de código 3.3: Archivo label_map.pbtxt

Luego de generar los archivos, se los va a subir al directorio */annotations* del repositorio de Github especificado en los parámetros del documento.

3.1.2.Entrenamiento

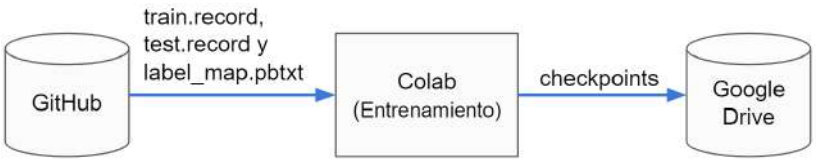


Figura 3.6: Componentes e información relacionados al entrenamiento

Input: archivos *train.record*, *test.record* y *label_map.pbtxt* del repositorio de GitHub.

Output: *checkpoints*, subidos a Google Drive.

La siguiente actividad de la plataforma es el entrenamiento del modelo de Machine Learning.

Una vez que se subieron los archivos *train.record*, *test.record* y *label_map.pbtxt* a Github se puede hacer uso de un nuevo documento de Google Colab para entrenar el modelo. El documento se encuentra en el anexo de códigos fuentes, bajo el nombre de "ENTRENAMIENTO.ipynb". Además, enhttp://mapo-lp.github.io/entrenamiento.htmlse encuentra una copia del mismo, lista para ser ejecutada en Google Colaboratory. El funcionamiento de este documento es el siguiente:
Primero se deben configurar ciertos parámetros en el documento, tal como el número de etapas de entrenamiento y la dirección del repositorio de Github que contiene los archivos generados por la actividad anterior. Además, se debe seleccionar el modelo pre entrenado a utilizar. Por defecto la plataforma selecciona el modelo "*ssd_mobilenet_v2_quantized*" ya que es una versión optimizada para correr en teléfonos celulares y que además funciona correctamente con cualquier problema que involucre objetos sencillos.

Para realizar el entrenamiento es necesario realizar algunas modificaciones en el archivo de configuración del modelo a reutilizar seleccionado. Este archivo puede ser

descargado desde el repositorio de TensorFlow[8] y posee toda la configuración a ser utilizada durante el entrenamiento. Como mínimo se deben modificar las rutas de los directorios donde se encuentran los archivos *train.record*, *test.record* y *label_map.pbtxt*, y además se debe modificar la cantidad de objetos a detectar. El archivo provee diversos parámetros (propios de TensorFlow), tal como el extractor de características, la tasa de aprendizaje y la función de *loss* que van a ser utilizadas para realizar el entrenamiento.

Al ejecutar el documento de Colab, el filesystem se enlazará con nuestra cuenta de Google Drive[9]. Este servicio de almacenamiento fue seleccionado ya que cuenta con una capacidad máxima (compartida con GMail y Google Photos) de 15GB[10] para su versión gratuita, lo cual nos va a permitir trabajar sin inconvenientes.
Luego se descargará el modelo pre entrenado a utilizar y se dará comienzo al entrenamiento, el cual se va a llevar a cabo utilizando la librería de código libre TensorFlow[11].

Durante el entrenamiento Colab mostrará diferente información relevante, tal como el *loss* y la métrica *mAP*, de los cuales ya hablamos en la sección2.2.3.
El valor del *loss* será mostrado al finalizar cada etapa del entrenamiento, como se observa en el fragmento de código3.4. Un valor normal para este indicador se encuentra entre 0 y 1, y si bien en las primeras etapas el valor es normalmente alto, el mismo va a ir disminuyendo a medida que transcurre el entrenamiento. Si el valor del loss no disminuye puede deberse a algún problema en el conjunto de imágenes. Si el valor se incrementa rápidamente puede ser como consecuencia de la aparición de *overfitting* en el modelo.

```
loss = 18.15808, step = 0
loss = 7.175391, step = 100
...
loss = 4.738452, step = 400
...
```

[8]https://github.com/tensorflow/models/tree/master/research/object_detection/samples/configs
[9]https://drive.google.com
[10]https://one.google.com/faq/storage
[11]https://www.tensorflow.org

```
loss = 1.3419428, step = 4100
...
loss = 0.211287, step = 13200
```

Fragmento de código 3.4: Resultados parciales del entrenamiento

En cuanto a la métrica mAP, la misma va a ser calculada periódicamente y va a ser mostrada de la siguiente manera (En este caso se muestran a modo de ejemplo las que contienen *maxDets=100*):

```
Average Precision @[ IoU=0.50:0.95 | area= all]   = 0.067
Average Precision @[ IoU=0.50      | area= all]   = 0.244
Average Precision @[ IoU=0.75      | area= all]   = 0.008
Average Precision @[ IoU=0.50:0.95 | area= small] = 0.023
Average Precision @[ IoU=0.50:0.95 | area=medium] = 0.087
Average Precision @[ IoU=0.50:0.95 | area= large] = 0.065
```

Fragmento de código 3.5: Informe de métrica mAP

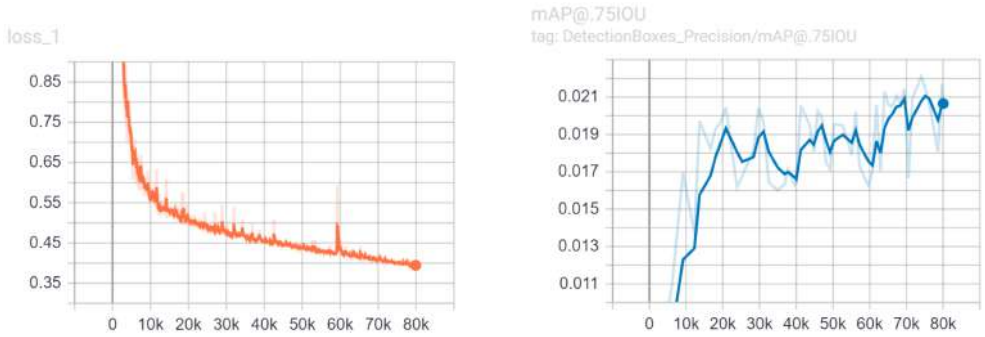
En el detalle de la métrica, se listan los resultados según el IoU (intersección sobre unión, descrita en la sección2.2.3), la cantidad máxima de detecciones y el tamaño de los cuadros delimitadores.

Si se desea visualizar un informe más detallado sobre el desempeño del modelo durante el entrenamiento se puede utilizar la herramienta TensorBoard[12], la cual viene incluida en el framework TensorFlow. Esta herramienta permite no solo observar el *loss* y el *mAP* (Figura3.7), sino también una gran cantidad de métricas diferentes y además, las predicciones realizadas en las distintas etapas de entrenamiento (Figura 3.8).

Durante el entrenamiento se van a realizar predicciones de manera periódica. El resultado de ellas puede encontrarse en la pestaña “images” de TensorBoard.

En la Figura3.8se pueden observar varios grupos de imágenes. En cada grupo se especifica la etapa en la cual se realizó la predicción. Allí, la imagen que se encuentra

[12]https://www.tensorflow.org/tensorboard



(a) informe de loss (b) informe de mAP

Figura 3.7: Informe de loss y mAP

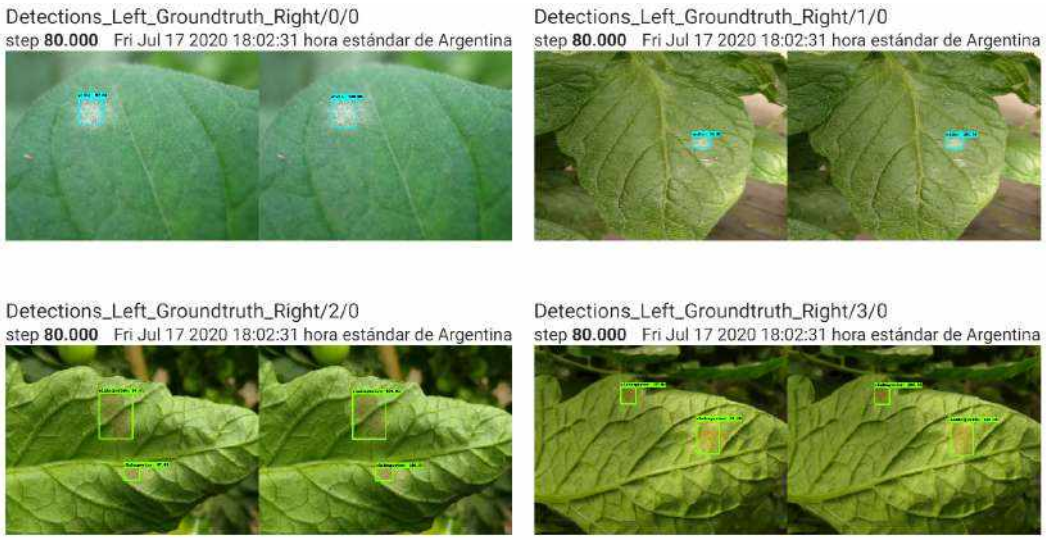


Figura 3.8: Predicciones en TensorBoard

a la derecha corresponde a la imagen original etiquetada y la imagen de la izquierda muestra la predicción realizada.

Como comentamos al inicio del capítulo, el entrenamiento es un proceso iterativo y de prueba y error. El usuario debe ir observando los resultados parciales devueltos por TensorFlow (y por TensorBoard si es posible) para ir realizando ajustes en el caso que los resultados en las predicciones y en las diferentes métricas no cumpla con las expectativas.

A medida que transcurre el entrenamiento se generarán diferentes checkpoints, los cuales guardan el estado (variables, operaciones, pesos) que contiene la red neuronal

en un determinado instante.

Los checkpoints serán subidos a Google Drive automáticamente, lo cual va a permitir generar el modelo TFLite y además, poder resumir el entrenamiento en caso que ocurra algún error.

3.1.3.Deploy del modelo TFLite

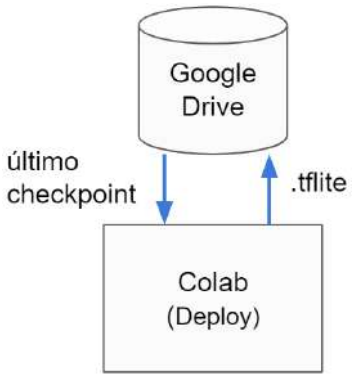


Figura 3.9: Componentes e información relacionados al deploy del modelo

Input: *checkpoints* en Google Drive.

Output: modelo *.tflite* subido a Google Drive.

Esta actividad tiene como objetivo generar el modelo TFLite a ser utilizado por la aplicación móvil.

En la aplicación se utiliza la librería *TensorFlow Lite*[13] para realizar la detección de objetos, la cual está diseñada para ejecutar modelos de manera eficiente en dispositivos que poseen recursos limitados. Parte de esa eficiencia proviene del uso del formato TFLite.

Los modelos generados por TensorFlow deben convertirse al formato TFLite para que *TensorFlow Lite* pueda hacer uso de ellos. Esta conversión reduce sustancialmente su tamaño e introduce optimizaciones que no afectan la precisión del mismo. Es posible, además, reducir aún más el tamaño del modelo y aumentar su velocidad de ejecución, a costa de alguna leve penalidad, como por ejemplo en su eficacia en la detección.

[13]https://www.tensorflow.org/lite

Para llevar a cabo esta actividad se usará un último documento de Colab, el cual se encuentra en el anexo de códigos fuentes, bajo el nombre de "GENERAR_MODELO_AUG.ipynb". Además, enhttp://mapo-lp.github.io/deploy.htmlse encuentra una copia del mismo, lista para ser ejecutada en Google Colaboratory. El funcionamiento de este documento es el siguiente:

Al ejecutar el documento, el filesystem se enlazará con nuestra cuenta de Google Drive, en donde se encuentran los checkpoints generados por la actividad anterior. El checkpoint a ser utilizado será el más reciente.

Para realizar la conversión del modelo se utilizarán dos scripts de TensorFlow (*tflite_convert* y *export_tflite_ssd_graph*), los cuales serán descargados automáticamente para su uso. El script "*tflite_convert*" permite convertir el modelo al formato TFLite. Este script opera con archivos con extensión ".pb", por lo que primero se debe convertir el checkpoint a ese formato.

La extensión *.pb* refiere a "*protobuf*", un tipo de archivo utilizado por TensorFlow que contiene la definición del grafo de la red neuronal y los pesos del modelo.

Para la generación del modelo .pb es necesario el script "*export_tflite_ssd_graph*", el cual se utiliza de la siguiente manera:

```
!python /content/models/research/object_detection/
   export_tflite_ssd_graph.py \
    --pipeline_config_path={pipeline_path} \
    --trained_checkpoint_prefix='{last_snapshot_path}' \
    --output_directory='{output_directory}' \
    --add_postprocessing_op=true
```

Fragmento de código 3.6: Generación del archivo .pb

Como se observa en el fragmento de código3.6, el script requiere que se le indique la ruta del último checkpoint generado, la ruta donde almacenar el archivo ".pb" y además, la ruta donde se encuentra el archivo de configuración del modelo, el cual se utilizó en la actividad anterior para entrenar la red.

```
!tflite_convert \
  --output_file='{output_directory}/detect.tflite' \
  --graph_def_file='{output_directory}/tflite_graph.pb' \
  --inference_type=QUANTIZED_UINT8 \
  --input_arrays='normalized_input_image_tensor' \
--output_arrays='TFLite_Detection_PostProcess,
   TFLite_Detection_PostProcess:1,TFLite_Detection_PostProcess:2,
   TFLite_Detection_PostProcess:3' \
  --mean_values=128 \
  --std_dev_values=128 \
  --input_shapes=1,300,300,3 \
  --change_concat_input_ranges=false \
  --allow_nudging_weights_to_use_fast_gemm_kernel=true
```

Fragmento de código 3.7: Generación del archivo detect.tflite

Al ejecutar el script se generará el archivo "*tflite_graph.pb*", el cual será utilizado en la ejecución del comando "*tflite_convert*", tal como se muestra en el fragmento de código 3.7.

La ejecución del script "*tflite_convert*" dará como resultado el archivo "*detect.tflite*", el cual será subido a Google Drive para su descarga desde la aplicación.

Para permitir que la aplicación pueda descargar un modelo y sus sucesivas actualizaciones utilizando la misma URL es necesario que la subida del archivo "*detect.tflite*" se realice siempre hacia la misma ubicación, con motivo de ir sobrescribiendo continuamente el mismo archivo. Eso permite, además, poder restaurar una versión anterior del modelo, en el caso que la nueva versión del modelo no sea tan efectiva como alguna anterior.

3.1.4.Uso del modelo en la Aplicación Móvil

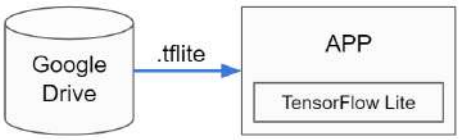
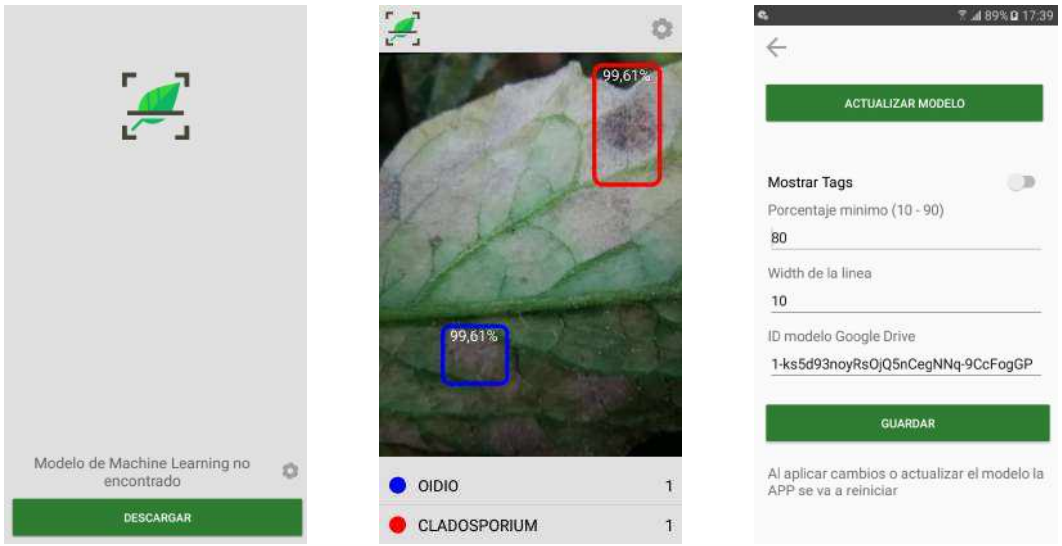


Figura 3.10: Componentes e información relacionados al uso del modelo

Input: modelo *.tflite* subido en Google Drive.

La última actividad de la plataforma es el uso del modelo detector de objetos a través de la aplicación móvil.

Para hacer uso del modelo en el teléfono celular se desarrolló una aplicación Android, cuyo código se encuentra en el anexo de códigos fuentes, bajo la carpeta "app_android". El código de la aplicación se encuentra además en el repositorio Git http://mapo-lp.github.io/android.html.
La aplicación requiere Android 5.0 o superior y tal como comentamos anteriormente hace uso de la librería TensorFlow Lite. Las pantallas que posee se muestran a continuación:



(a) pantalla inicial (b) pantalla de cámara (c) pantalla de configuración

Figura 3.11: Pantallas de la aplicación móvil

Pantalla Inicial Esta pantalla es mostrada cuando no existe un modelo de Machine Learning descargado. Permite realizar la descarga del mismo y además, provee un enlace a la pantalla de Configuración, de la cual vamos a hablar luego. En esta pantalla también se controla el permiso de acceso a la cámara, el cual debe ser aceptado por el usuario para poder utilizar la aplicación.

Pantalla de Cámara En esta pantalla se realiza la detección de objetos en tiempo real. Allí se muestra un “preview” de la cámara trasera del teléfono en donde se indican

los objetos detectados. Para el manejo de la cámara se hace uso de la API `Camera2`[14] de Android.

Luego de enlazar la librería de TensorFlow Lite con el modelo descargado y de configurar el uso de la cámara trasera, se comienza con el ciclo de detección, el cual sigue los siguientes pasos:

1. Cada algunos milisegundos el sistema obtiene una nueva imagen desde la cámara e invoca a un callback para informar sobre la existencia de la misma. El callback invocado depende del hardware del teléfono celular. Si el hardware posee soporte total de la API de `Camera2` se invoca al método

   ```
   public void onImageAvailable(final ImageReader reader)
   ```

   caso contrario se invoca al método

   ```
   public abstract void onPreviewFrame (byte[] data, Camera camera)
   ```

2. Dentro del callback, se procesa la imagen y se le hacen ciertas transformaciones, entre las que se incluye la conversión a `Bitmap` para poder ser manejada por la librería de TensorFlow Lite. Luego se la envía al método

   ```
   List<Recognition> recognizeImage(Bitmap bitmap)
   ```

   de la clase `Classifier`, el cual recibe un `Bitmap`, ejecuta la detección de objetos sobre él y retorna un listado con los objetos detectados (de clase `Recognition`). La clase `Recognition` posee el id, nombre, nivel de confianza y cuadro delimitador de cada objeto detectado.
   Es importante aclarar que las clases `Classifier` y `Recognition` forman parte de la librería de TensorFlow Lite.

[14]https://developer.android.com/reference/android/hardware/camera2/package-summary

3. Una vez que se devuelven los objetos detectados, serán desestimados los que no cumplen con el mínimo de confianza establecido por el usuario. Este valor por defecto es de 80 %, pero puede ser modificado desde la pantalla de Configuración. Luego se muestran las detecciones al usuario (como se muestra en la Figura3.11b) y se vuelve al Punto1para reiniciar el ciclo de detección.

Como se observa en la Figura3.11b, los objetos detectados son marcados con un rectángulo que indica la posición, confianza en la predicción y opcionalmente el nombre de cada uno de ellos. Cada clase de objeto posee un color generado aleatoriamente y es posible detectar múltiples instancias de múltiples clases a la vez. Además de mostrar los objetos en la imagen de la cámara, se los lista debajo de la misma, en donde se indica el nombre, color y cantidad de instancias detectadas del mismo.

Pantalla de Configuración Esta pantalla le brinda al usuario la posibilidad de actualizar el modelo de Machine Learning. La ruta desde donde se descarga el modelo puede ser modificada desde la misma pantalla.
Además, permite la modificación de algunas características de la aplicación, tal como el grosor del rectángulo utilizado para marcar los objetos, el porcentaje mínimo de confianza utilizado para filtrar los resultados de las predicciones y si se desea mostrar o no el nombre de los objetos detectados. Un porcentaje de confianza mínimo de 80 % indica que si se devuelven predicciones con un 30 o 40 % de confianza, las mismas serán desestimadas.

3.2.Uso de la plataforma para la detección de enfermedades y plagas en cultivos

En esta sección describiremos cómo se utilizó la plataforma para entrenar un modelo capaz de detectar síntomas de dos enfermedades del tomate, lo que demuestra que es posible hacer uso de la misma para la detección de síntomas en cualquier tipo de

cultivo, con el fin de proveer una herramienta que ayude a la toma de decisiones en el agro.

A continuación iremos comentando el trabajo realizado y los resultados obtenidos.

Para el armado del conjunto de imágenes de entrenamiento contamos con la colaboración de personal de la Facultad de Agronomía de la Universidad Nacional de La Plata, al cual agradecemos ya que nos brindó una gran cantidad de imágenes de tomate con síntomas de dos de sus enfermedades: cladosporium y oidio.

El cladosporium [37] es una enfermedad que se desarrolla en condiciones de humedad superiores a 70 % y temperatura entre 5-25º C. Sus síntomas se presentan como manchas de color amarillo pálido en el haz (cara de las hojas que se expone al sol) y como un moho de color gris o pardo en el envés (reverso de la hoja).

Figura 3.12: Daño causado por cladosporium

El odio [38], por su parte, es una de las enfermedades más comunes en el tomate. Se trata de un hongo que actúa sobre los tallos y hojas de la planta. Las condiciones óptimas para su desarrollo son una temperatura entre 20-25º C y una humedad de 50-70 %. Los síntomas iniciales consisten en manchas blancas y pulverulentas en el haz que se van tornando de color amarillo y detrás de las cuales puede verse un polvillo blanquecino en el envés. Estas manchas pueden crecer y extenderse afectando al desarrollo vital de toda la planta.

Figura 3.13: Daño causado por oídio

La cantidad de imágenes utilizada fue de 146, 49 de las cuales correspondían a cladosporium, 43 a oidio y las restantes 54 no presentaban síntomas de ninguna de las dos enfermedades. Estas últimas imágenes fueron agregadas para reducir el posible overfitting y los falsos positivos. Entre ellas se encontraban imágenes de plantas sanas, imágenes que tenían síntomas de otras enfermedades y algunas imágenes aleatorias descargadas de internet.

Luego de etiquetar las imágenes con LabelImg se las dividió en los conjuntos *training* y *test*, con una distribución del 70 y 30 % respectivamente y se las subió a Github.

A continuación se realizó el aumento de las imágenes utilizando el documento de Colab respectivo. Como disponíamos de tan solo 146 imágenes en total, era necesario generar una gran cantidad de imágenes extras, para permitirle a la red neuronal un mejor aprendizaje. Se optó por generar 30 imágenes para cada imagen que poseía síntomas etiquetados, lo que resultó en un total de 2906 imágenes para el entrenamiento. Las transformaciones utilizadas fueron las siguientes:

```
aug = iaa.SomeOf(2, [
    iaa.Affine(scale=(0.5, 1.5)),
    iaa.Affine(rotate=(-60, 60)),
    iaa.Affine(translate_percent={"x":(-0.3, 0.3),"y":(-0.3, 0.3)}),
    iaa.Fliplr(1),
    iaa.GaussianBlur(sigma=(1.0, 3.0)),
```

```
    iaa.AdditiveGaussianNoise(scale=(0.03*255, 0.05*255))
])
```

Fragmento de código 3.8: Transformaciones utilizadas en el entrenamiento

A continuación pueden observarse algunas de las transformaciones realizadas a una imagen en particular.

Figura 3.14: Transformaciones realizadas a una imagen

Luego de realizar el aumento, el documento de Colab generó los archivos *train.record*, *test.record* y *label_map.pbtxt*, los cuales subió al repositorio de Github.

Luego se continuó con el documento de Colab encargado de entrenar la red neuronal. Se entrenó el modelo por un total de 80.000 etapas de entrenamiento utilizando el modelo pre entrenado "*faster_rcnn_inception_v2*". En el archivo de configuración del modelo[15] sólo se modificó la ruta de los directorios y la cantidad de objetos a detectar, la configuración restante se dejó intacta.

El entrenamiento se llevó a cabo durante **16h:20m:36s**. En la figura3.15puede observarse el horario de inicio y el horario de fin. El tiempo relativo (*relative*) se refiere al tiempo en que efectivamente se estuvo entrenando el modelo (el entrenamiento

[15]https://github.com/tensorflow/models/blob/master/research/object_detection/samples/configs/faster_rcnn_inception_v2_coco.config

fue interrumpido durante la noche del 18 de Julio y resumido en la mañana del día siguiente).

Name	Smoothed	Value	Step	Time	Relative
.	2e-4	2e-4	1	Sat Jul 18, 23:19:09	0s

(a) horario de inicio

Name	Smoothed	Value	Step	Time	Relative
.	0.1854	0.2204	79.9k	Sun Jul 19, 15:39:46	16h 20m 36s

(b) horario de finalización

Figura 3.15: Horario de inicio y finalización del entrenamiento

El resultado de Precisión y Recall obtenidos en la ultima evaluación (etapa 80.000) es mostrado en la figura3.16.

```
Running per image evaluation...
Evaluate annotation type *bbox*
DONE (t=1.98s).
Accumulating evaluation results...
DONE (t=0.36s).
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.692
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.958
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.791
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.516
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.730
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.710
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.321
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.732
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.733
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.542
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.772
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.721
INFO:tensorflow:Finished evaluation at 2020-07-19-18:40:47
```

Figura 3.16: Evaluación final

Como comentamos durante el desarrollo del trabajo, Tensor Board provee una interfaz muy interesante que permite observar el desempeño de la red a través de las distintas etapas del entrenamiento. Es posible visualizar diferentes métricas, tal como el *mAP* (Figura3.17) y el *mAR* (Figura3.18), la cual es una métrica que calcula el *Recall* de las predicciones.

Otro apartado interesante en Tensor Board es el de las predicciones realizadas a las imágenes (Figura3.19), en donde se compara la imagen etiquetada con el resultado de la predicción en cierta etapa (en este caso la final).

Luego de finalizar el entrenamiento se utilizó el último documento de Colab para generar el modelo *TFLite*. Con el modelo subido a Google Drive, se lo descargó desde la aplicación móvil.

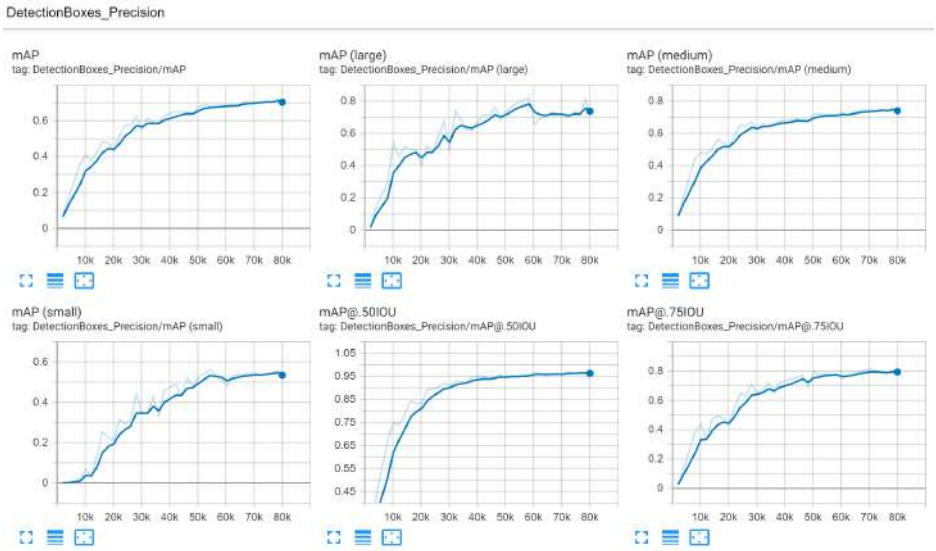


Figura 3.17: Precisión final

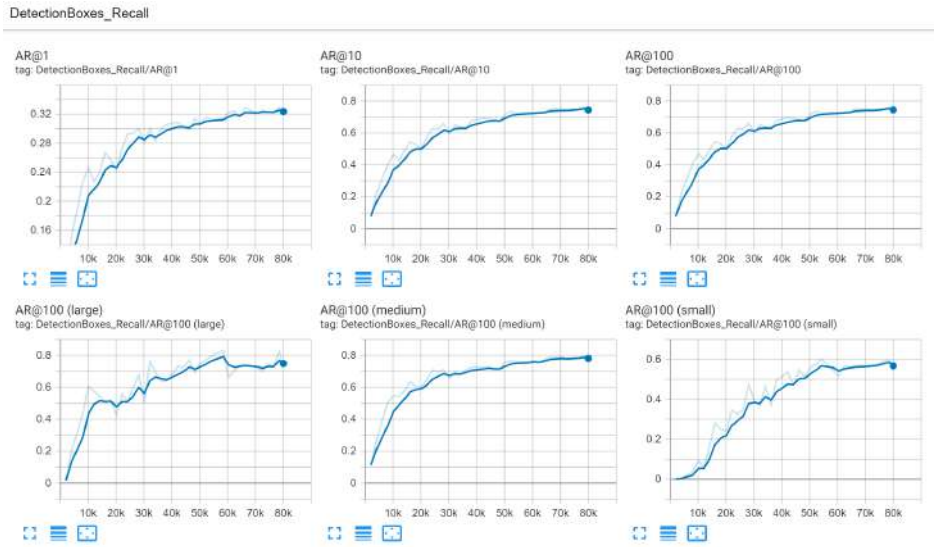


Figura 3.18: Recall final

En las imágenes de la Figura3.20se puede apreciar las diferentes detecciones realizadas por la aplicación.

Figura 3.19: Predicciones realizadas por TensorFlow

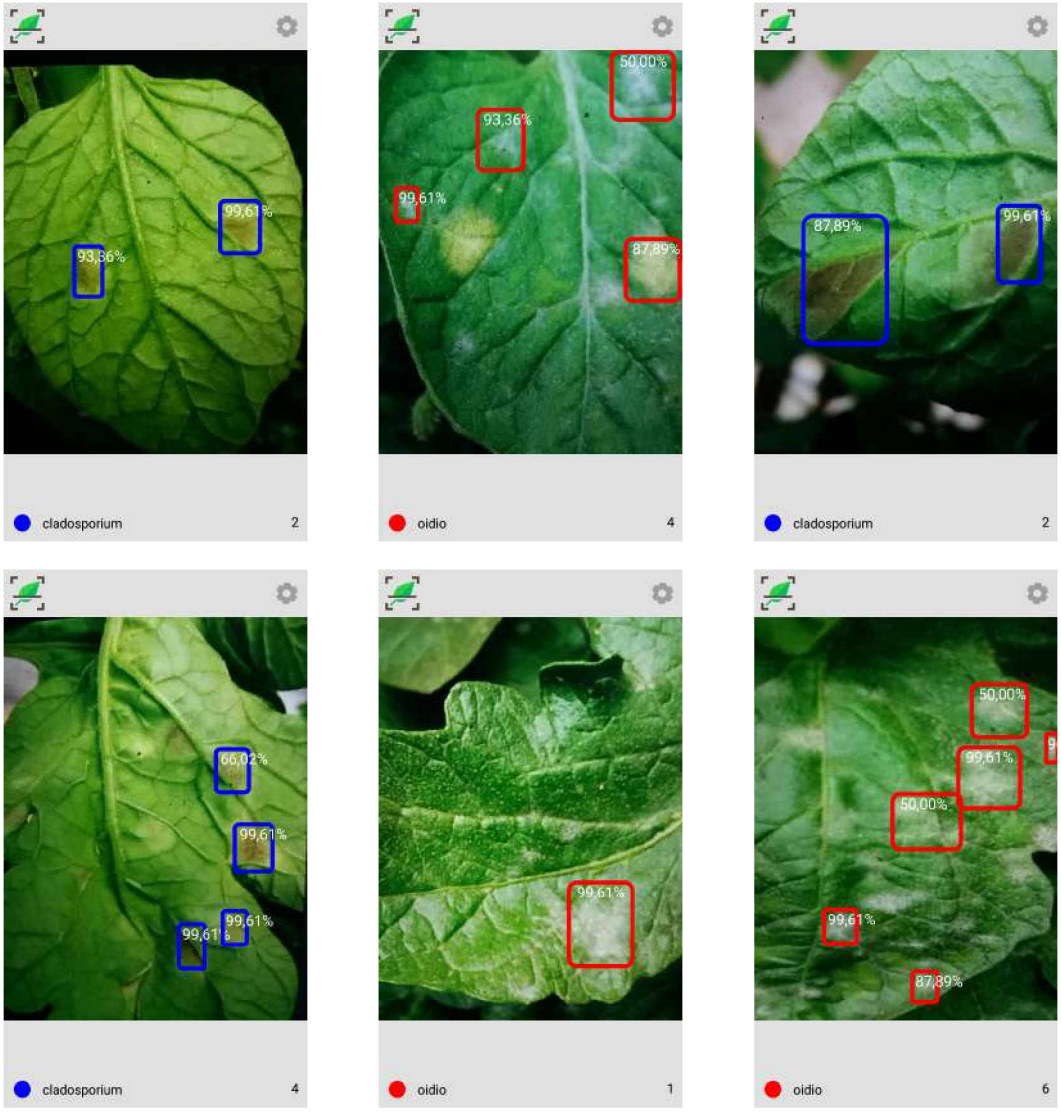


Figura 3.20: Predicciones realizadas por la aplicación

4. Conclusiones

El presente trabajo de investigación no pretende profundizar sobre el campo de Machine Learning ya que existe amplia literatura sobre el mismo; por el contrario, pretende apalancar una solución integral a partir de la composición de soluciones existentes disponibles en la nube.

La plataforma presentada fue utilizada para la detección de las enfermedades oidio y cladosporium en el tomate. Los resultados en la utilización de la plataforma para llevar a cabo dicha tarea fueron más que satisfactorios, ya que logró detectar de manera correcta los síntomas. Eso demuestra que la plataforma puede ser utilizada en el sector de la agricultura, como una herramienta adicional para la detección de enfermedades y plagas con el fin de combatir la problemática y de reducir sus consecuencias.

Si bien la plataforma fue concebida para lo mencionado anteriormente, desde el primer momento se tuvo en mente generar una solución integral, genérica, configurable, gratuita y fácil de utilizar, para que la misma pueda ser aplicada a cualquier dominio de interés y pueda ser utilizada por cualquier persona que desee hacerlo.

Algo que creemos es la principal diferencia frente a investigaciones similares de detección de síntomas es el uso del aumento de imágenes. En general las investigaciones relacionadas a plantas utilizan el dataset de Plant Village [13], el cual es un gran dataset que cuenta con más de 50.000 imágenes. El principal problema de entrenar modelos utilizando ese dataset es que las imágenes que contiene son imágenes obtenidas bajo condiciones controladas, lo que conlleva a que la red neuronal se entrene con imágenes "perfectas", en vez de con imágenes del "mundo real". Eso genera que cuando el modelo deba inferir imágenes reales su precisión pueda verse ampliamente perjudicada.

Al día de hoy no hemos encontrado una plataforma similar a la propuesta en este trabajo y por ello queremos dedicar unas líneas a la importancia de hacer accesible

una tecnología para que todas las personas puedan aprender sobre ella y utilizarla.
El acceso de las personas a las tecnologías es fundamental y debe ser universal. A través de este trabajo intentamos contribuir a ese objetivo. La plataforma posibilita el acceso a través de los siguientes puntos: primero, el uso de servicios gratuitos permite que alguien que no puede pagar por un servicio pueda hacer uso total de la plataforma. Segundo, el uso de hardware potente (y gratuito) vía Google Colab posibilita que personas que cuentan con computadoras regulares puedan entrenar grandes modelos detectores de objetos. Tercero, la plataforma fue concebida, entre otras cosas, para que las personas puedan acceder al Machine Learning de una manera sencilla y didáctica. En las primeras etapas de investigación llevada a cabo para realizar este trabajo se observó que la curva de aprendizaje para lograr entrenar y usar redes neuronales resulta elevada. Ello se da principalmente porque la gran mayoría de la documentación se encuentra dispersa y en inglés y porque no hay procedimientos que guíen sobre como realizar desde el entrenamiento hasta el uso de un modelo en una aplicación móvil. En consecuencia, se requiere mucho tiempo para aprender todo lo que esta temática involucra.
La plataforma pretende ser utilizada como un “paso a paso” y por ello fue “separada” en sencillas actividades conectadas entre si, con motivo de guiar a través de todo el proceso que la misma contempla a quien desee aprender sobre este tema.

Se espera que el sistema propuesto haga una contribución significativa en las áreas en donde se lo utilice y que sea un recurso valioso de aprendizaje para todo aquel que desee iniciarse en el mundo del Machine Learning.

5. Trabajos Futuros

Debido a las características de la plataforma desarrollada, los trabajos futuros a ser aplicados sobre la misma son de los más variados y dependen del dominio de interés en donde desee utilizarse. Se proponen las siguientes actividades:

- Prueba de campo del detector de síntomas en cultivos

  Debido a la actual pandemia de Covid-19 que estamos atravesando nos fue imposible realizar una prueba de campo del detector de síntomas de enfermedades y plagas en cultivos. Esta actividad era algo clave y estaba planeada desde que se arrancó con la investigación, ya que nos iba a permitir obtener un valioso feedback para poder mejorar la plataforma en general y el detector en particular.

  Actualmente estamos trabajando junto al Director de la presente Tesis en la instrucción de personas que van a darle continuidad al proyecto.

- Uso de la plataforma en diversos dominios

  Aprovechando que la plataforma es genérica y que puede ser utilizada para detectar cualquier tipo de objeto en imágenes, sería interesante hacer uso de la misma en diferentes dominios, lo cual va a servir además para evaluar su comportamiento y para realizar ajustes, para así mejorar y optimizar la plataforma en general. Un campo en donde podría ser utilizada es en medicina, para evaluar tomografías o radiografías en la búsqueda de diferentes anomalías por ejemplo.

- Reemplazo de LabelImg por una herramienta cloud

  Tal como mencionamos durante el desarrollo del trabajo, el único componente no cloud utilizado es LabelImg, el cual realiza el etiquetado de las imágenes. Al momento de su uso no hallamos una herramienta cloud gratuita que funcione igual o mejor que ella. Las herramientas utilizadas presentaban diferentes in-

convenientes, tal como la imposibilidad de poder cargar una gran cantidad de imágenes a ser etiquetadas. Sería ideal entonces encontrar una herramienta cloud para esa actividad, para así contar con el 100 % de los componentes cloud.

- Migración a TensorFlow 2

  Cuando se comenzó a desarrollar el trabajo la versión de TensorFlow utilizada en Google Colab era la 1.15. Actualmente Colab migró a la versión 2.0, aunque es posible continuar utilizando la versión anterior, por lo que nuestro código sigue funcionando correctamente. Se deja como actividad futura la adaptación del código de los diferentes documentos de Colab a la nueva versión de TensorFlow para así poder aprovechar todas sus mejoras y nuevas funcionalidades.

- Optimización de los parámetros

  Considerando que la plataforma cuenta con múltiples parámetros en cada una de sus etapas, un posible trabajo futuro sería trabajar lograr encontrar los mejores valores para ellos que conduzcan a que el modelo generado sea lo más eficaz posible para cualquier tipo de problema.

- Posibilidad de crear redes neuronales desde cero (*from Scratch*)

  Uno de los objetivos de la plataforma es permitir que cualquier persona sin mucha experiencia en el campo de Machine Learning sea capaz de entrenar un detector de objetos de manera sencilla. Podría evaluarse la posibilidad de realizar algunas modificaciones en la plataforma para permitir la codificación de redes neuronales propias, para permitir entrenar modelos que requieran ser más específicos o complejos.

- Hacer uso del modelo en otras plataformas

  El modelo generado por nuestra plataforma de Machine Learning es un archivo con formato *.tflite*, el cual es requerido por la librería TensorFlow Lite para ejecutar el modelo en diferentes plataformas. Si bien la utilizamos en Android, es posible utilizarla también en iOS, en plataformas Linux (por ejemplo en Raspberry

Pi[1]) y en microcontroladores como Arduino[2]. Sería interesante entonces realizar experimentos en esas plataformas.

- Enriquecer la aplicación móvil

  La aplicación móvil podría incorporar diferentes opciones y funcionalidades, tal como el agregado de un apartado con información sobre los diferentes objetos que se pueden reconocer y la configuración de un sistema de alertas que se dispare al detectar un objeto en particular.

  Se podría añadir la posibilidad de soportar múltiples modelos, lo cual es una modificación fácil de realizar, y consistiría en hacer el deploy de 2 modelos diferentes por ejemplo. Entonces, el usuario podría cambiar entre un modelo y otro con solo apretar un botón.

  Por último, si la aplicación fuera a ser utilizada en el campo de la educación, se podría considerar el agregado de diferentes componentes de Gamificación a la misma, para lograr un aprendizaje más didáctico.

[1]https://www.raspberrypi.org
[2]https://www.arduino.cc

Bibliografía

[1] Sumit Saha. A comprehensive guide to convolutional neural networks — the eli5 way.https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, 2018. [Online; accessed 20-July-2020].

[2] FAO-PESA Centroamérica. Seguridad alimentaria nutricional, conceptos básicos. http://www.fao.org/3/a-at772s.pdf, 2011. [Online; accessed 20-July-2020].

[3] International Plant Protection Convention FAO. Plant health and food security. http://www.fao.org/3/a-i7829e.pdf, 2017. [Online; accessed 20-July-2020].

[4] FAO, FIDA, OMS, PMA, and UNICEF. El estado de la seguridad alimentaria y la nutrición en el mundo 2019. protegerse frente a la desaceleración y el debilitamiento de la economía. Roma, Italia, 2019.

[5] FAO. La agricultura mundial en la perspectiva del año 2050. *Cómo alimentar al mundo en 2050*, 2009.

[6] FAO. Año internacional de la sanidad vegetal. Roma, Italia, 2019. (CA6992ES/1/11.19).

[7]FAO. *El estado mundial de la agricultura y la alimentación*. 2015.

[8] Instituto Nacional de Tecnología Agropecuaria. La agricultura familiar produce casi el 80 por ciento de los alimentos.https://inta.gob.ar/noticias/la-agricultura-familiar-produce-casi-el-80-por-ciento-de-los-alimentos, 2017. [Online; accessed 20-July-2020].

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[10] S. O'Dea. Number of smartphone users worldwide from 2016 to 2021.https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide, 2020.

[11] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3):210–229, July 1959.

[12] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., USA, 1 edition, 1997.

[13] David Hughes, Marcel Salathé, et al. An open access repository of images on plant health to enable the development of mobile disease diagnostics. *arXiv preprint arXiv:1511.08060*, 2015.

[14] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, December 1989.

[15] Jihen Amara, Bassem Bouaziz, and Alsayed Algergawy. A deep learning-based approach for banana leaf diseases classification. In *BTW*, 2017.

[16] Sharada Mohanty, David Hughes, and Marcel Salathe. Using deep learning for image-based plant disease detection. *Frontiers in Plant Science*, 7, 04 2016.

[17] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[18] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[19] Guan Wang, Yu Sun, and Jianxin Wang. Automatic image-based plant disease severity estimation using deep learning. *Computational Intelligence and Neuroscience*, 2017:1–8, 07 2017.

[20] M. Islam, Anh Dinh, K. Wahid, and P. Bhowmik. Detection of potato diseases using image segmentation and multiclass support vector machine. In *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–4, 2017.

[21] Alvaro Fuentes, Sook Yoon, Sang Cheol Kim, and Dong Sun Park. A robust deep-learning-based detector for real-time tomato plant diseases and pests recognition. *Sensors (Basel, Switzerland)*, 17, 2017.

[22] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, 2017.

[23] Muhammad Waseem Tahir, Nayyer Abbas Zaidi, Adeel Akhtar Rao, Roland Blank, Michael J. Vellekoop, and Walter Lang. A fungus spores dataset and a convolutional neural network based approach for fungus detection. *IEEE Transactions on NanoBioscience*, 17:281–290, 2018.

[24] Adhao Asmita Sarangdhar and V. R. Pawar. Machine learning regression technique for cotton leaf disease detection and controlling using iot. *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, 2:449–454, 2017.

[25] Stacey Ronaghan. Deep learning: Which loss and activation functions should i use?https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8, 2018. [Online; accessed 20-July-2020].

[26] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

[27]Kunihiko Fukushima. Neocognitron. *Scholarpedia*, 2(1):1717, January 2007.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[29] ImageNet. Large scale visual recognition challenge 2012 (ilsvrc2012).http://www.image-net.org/challenges/LSVRC/2012/results.html, 2012.

[30] Wikipedia contributors. Vanishing gradient problem — Wikipedia, the free encyclopedia. [Online; accessed 20-July-2020].

[31] Jason Brownlee. How to fix the vanishing gradients problem using the relu.https://machinelearningmastery.com/how-to-fix-vanishing-gradients-using-the-rectified-linear-activation-function, 2019. [Online; accessed 20-July-2020].

[32] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *ArXiv*, abs/1704.04861, 2017.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[34] W Liu, D Anguelov, D Erhan, C Szegedy, S Reed, CY Fu, and AC Berg. Ssd: Single shot multibox detector. arxiv 2016. *arXiv preprint arXiv:1512.02325*, 2020.

[35] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[36] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[37] Cladosporium.https://www.syngenta.es/cultivos/tomate/enfermedades/cladosporiosis. [Online; accessed 20-July-2020].

[38] Oidio.https://www.syngenta.es/cultivos/tomate/enfermedades/oidiopsis. [Online; accessed 20-July-2020].