



**EDUCACIÓN**  
SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNOLÓGICO  
NACIONAL DE MÉXICO



---

**Instituto Tecnológico de Chihuahua II**  
DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

**CLASIFICADOR VERSÁTIL DE PRODUCTOS AGRÍCOLAS  
POR MEDIO DE DEEP LEARNING**

TESIS

PARA OBTENER EL GRADO DE

**MAESTRO EN SISTEMAS COMPUTACIONALES**

PRESENTA

**LUIS BRYAN MARTINEZ LOZOYA**

DIRECTOR DE TESIS

DR. HERNÁN DE LA GARZA GUTIÉRREZ

CODIRECTOR DE TESIS

M.C. ARTURO LEGARDA SÁENZ

---

CHIHUAHUA, CHIH., JUNIO 2023

# Dictamen

Chihuahua, Chihuahua, 02 de junio 2023

**M.C. MARIA ELENA MARTINEZ CASTELLANOS**

**COORDINADORA DE POSGRADO E INVESTIGACION.**

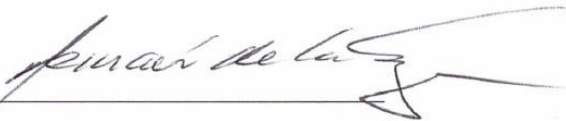
**PRESENTE**

Por medio de este conducto el comité tutorial revisor de la tesis para obtención de grado de Maestro en Sistemas Computacionales, que lleva el nombre de: "CLASIFICADOR VERSÁTIL DE PRODUCTOS AGRICOLAS POR MEDIO DE DEEP LEARNING" que presenta el C. LUIS BRYAN MARTINEZ LOZOYA, hace de su conocimiento que después de ser revisado ha dictaminado la APROBACIÓN de la misma.

Sin otro particular de momento queda de usted.

Atentamente

La Comisión de Revisión de Tesis.



---

DR. HERNÁN DE LA GARZA GUTIERREZ

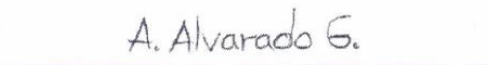
Director de tesis



---

M.C. ARTURO LEGARDA SÁENZ

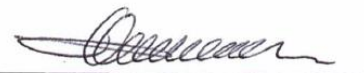
Co –Director



---

M.I.S.C. JESUS ARTURO ALVARADO GRANADINO

Revisor



---

DR. ALBERTO CAMACHO RIOS

Revisor

## RESUMEN

El asegurar el abasto de alimentos para los próximos años será un gran reto. Uno de los desafíos de la agricultura, se centra en incrementar la productividad para satisfacer las necesidades de una población que continúa en aumento. En el presente documento se propone el empleo de Deep Learning (Aprendizaje profundo), para apoyar a las pequeñas y medianas empresas mexicanas del sector agrícola. Haciendo uso del algoritmo de detección en tiempo real YOLO (you only look once), en específico los modelos YOLOv4, YOLOR y YOLOX, se realizaron pruebas para comparar su precisión y velocidad en la detección de cuatro productos agrícolas: manzana, plátano, naranja y chile jalapeño. Los tres modelos lograron detecciones notablemente rápidas con una media de precisión mayor al 91%, cualidades que los hacen llamativos para ser utilizados en los procesos de control de calidad. Por último, empleando el modelo YOLOv4, se realizó una aplicación de escritorio capaz de detectar productos en imágenes y en tiempo real por medio de una cámara web.

## **ABSTRACT**

Ensuring food supply for the coming years will be a great challenge. One of the challenges in agriculture is focused on increasing productivity to meet the needs of a growing population. In this present document, the use of Deep Learning is proposed to support Mexican small and medium-sized companies in the agricultural sector. By employing the real-time detection algorithm YOLO (you only look once), specifically the models YOLOv4, YOLOR and YOLOX, tests were conducted to compare their accuracy and speed in detecting four agricultural products: apple, banana, orange and jalapeño pepper. All three models achieved remarkably fast detections with an average precision above 91%, qualities that make them appealing for use in quality control processes. Lastly, using the YOLOv4 model, a desktop application was developed capable of detecting products in images and in real-time using a webcam.

# CONTENIDO

I. INTRODUCCIÓN	1
1.1 Introducción	1
1.2 Planteamiento del problema.	2
1.3 Alcances y limitaciones	2
1.4 Justificación	3
1.5 Objetivo	3
1.5.1 Objetivo general	3
1.5.2 Objetivo específico	3
1.6 Hipótesis	4
II. ESTADO DEL ARTE	5
III. MARCO TEÓRICO	8
3.1 Lenguaje de programación Python	8
3.2 OpenCV	9
3.3 TensorFlow	9
3.4 Keras	10
3.5 Deep learning	11
3.6 Metodología Kanban	14
3.6.1 Principios	14
3.6.2 Prácticas	15
3.7 Algoritmo de detección	17
3.7.1 YOLO (you only look once)	18
3.7.2 YOLOv4 arquitectura	21
3.8 Transfer learning	22
3.9 Data augmentation	23
3.10 Overfitting	23
3.11 Algoritmo de optimización	23
IV. DESARROLLO	26
4.1 Análisis de software	26
4.1.1 Requisitos funcionales y no funcionales	26

4.1.2 Diseño de la aplicación	26
4.2 Desarrollo de software	29
4.2.1 Entrenamiento modelo de red neuronal	29
4.2.1.1 Primer acercamiento: Clasificación de objetos mediante Keras	30
4.2.1.2 Segundo acercamiento: Algoritmo YOLO	37
4.2.2 Módulo detección en tiempo real	44
4.2.3 Módulo detección en imagen	50
V. RESULTADOS	51
5.1 Resultados entorno Colab	51
5.2 Detecciones en la aplicación	52
VI. CONCLUSIONES	55
VII. BIBLIOGRAFÍA	56

## INDICE DE FIGURAS

Figura 3.1 Tablero Kanban (kanbanize, 2022).	14
Figura 3.2. Chile jalapeño. Objeto a detectar.	18
Figura 3.3. Múltiples detecciones en un mismo objeto.	19
Figura 3.4. Detecciones restantes tras la aplicación de la técnica IoU.	20
Figura 3.5. Detección restante tras la aplicación de la técnica non-max suppression.	20
Figura 3.6. Arquitectura algoritmos de detección (Bochkovskiy et al, 2020).	211
Figura 4.1. Diseño de la interfaz gráfica.	277
Figura 4.2. Diagrama caso de uso.	277
Figura 4.3. Diagrama de componentes y distribución.	288
Figura 4.4. Pimiento morrón podrido (Potdar, 2021).	30
Figura 4.5. Pimiento morrón fresco (Potdar, 2021).	30
Figura 4.6. Preprocesamiento de la información.	311
Figura 4.7. Función ModelCheckpoint.	311
Figura 4.8. Modelo secuencial.	322
Figura 4.9. Modelo base.	322
Figura 4.10. Creación capa de entrada.	322
Figura 4.11. Capa densa y dropout.	333
Figura 4.12. Capa AveragePooling y de salida. Modelo final.	333
Figura 4.13. Compilación del modelo.	333
Figura 4.14. Entrenamiento del modelo.	344
Figura 4.15. Evaluación del modelo.	344
Figura 4.16. Utilización de la técnica fine-tuning.	355
Figura 4.17. Compilación del modelo, etapa fine-tuning.	355
Figura 4.18. Entrenamiento del modelo, etapa fine-tuning.	355
Figura 4.19. Evaluación del modelo, etapa fine-tuning.	355
Figura 4.20. Resultados del modelo. Librería Keras.	366
Figura 4.21. Predicción acertada del modelo. Librería Keras.	377
Figura 4.22. Repositorio darknet.	399
Figura 4.23. Establecer conexión con Google Drive.	399
Figura 4.24. Contenido del archivo .names.	40
Figura 4.25. Fragmento del contenido del archivo train.txt. Donde se indican las rutas de las imágenes.	40
Figura 4.26. Contenido del archivo .data.	41
Figura 4.27. Archivo de configuración. Dimensiones.	41
Figura 4.28. Archivo de configuración. Variable max_batches.	41
Figura 4.29. Archivo de configuración. Variable steps.	411
Figura 4.30. Archivo de configuración. Capa yolo.	422
Figura 4.31. Archivo de configuración. Capa de convolución.	422
Figura 4.32. Entrenamiento del modelo YOLOv4.	422
Figura 4.33. Resultados de las iteraciones del entrenamiento.	433
Figura 4.34. Librerías para el desarrollo del módulo de detección.	455
Figura 4.35. Creación de un objeto VideoCapture.	455

Figura 4.36. Procesamiento del archivo .names. ....	455
Figura 4.37. Creación de la red neuronal mediante OpenCV.....	455
Figura 4.38. Nombre de las capas de la red neuronal.....	466
Figura 4.39. Obtener capas de salida de la red neuronal.....	466
Figura 4.40. Probabilidad mínima para eliminar predicciones débiles. ....	466
Figura 4.41. Umbral utilizado en la técnica de non-max suppression.....	466
Figura 4.42. Capturar frames de la cámara.....	466
Figura 4.43. Obtener blob del frame. ....	477
Figura 4.44. Instrucciones para la detección de objetos.....	477
Figura 4.45. Obtener probabilidad más alta del objeto detectado.....	488
Figura 4.46. Creación del cuadro delimitador.....	488
Figura 4.47. Obtener coordenadas del cuadro delimitador.....	488
Figura 4.48. Aplicación de la técnica Non-max suppression.....	49
Figura 4.49. Dibujar cuadro delimitador.....	49
Figura 4.50. Asignar texto junto cuadro delimitador.....	49
Figura 4.51. Mostrar las detecciones en una ventana.....	50
Figura 4.52. Instrucción para leer la imagen a detectar.....	50
Figura 4.53. Instrucción para guardar la detección.....	50
Figura 5.1. Detecciones con modelo YOLOX.....	51
Figura 5.2. Detecciones con modelo YOLOv4.....	51
Figura 5.3. Detecciones con modelo YOLOR.....	51
Figura 5.4. Resultados de los modelos.....	52
Figura 5.5. Interfaz gráfica.....	52
Figura 5.6. Detección en imagen.....	533
Figura 5.7. Detección en imagen. Varios objetos, misma clase.....	533
Figura 5.8. Detección en imagen. Varios objetos, distintas clases.....	544



### I. INTRODUCCIÓN

#### 1.1 Introducción

Uno de los desafíos fundamentales de la agricultura, se centra en mejorar su productividad para una población que continúa aumentando. Se requiere aumentar la productividad en un 50% para cubrir la demanda creciente para el año 2050 (FAO (Food and Agriculture Organization), 2017).

Asegurar el abasto de alimentos para los próximos años en México será un gran reto, porque aún y cuando el área cultivable es de aproximadamente 27 millones de hectáreas (INEGI-SAGARPA, 2015), esta superficie podría disminuir, debido a que se puede cambiar su uso agrícola para uso urbano. En ocasiones, el incremento de la producción alimentaria y el crecimiento económico se han conseguido a costa del medio ambiente natural (FAO, 2017), opción que resulta poco viable.

En varios países de Latinoamérica el aumento de la producción agrícola es atribuido a la mejora realizada en la productividad de los cultivos (FAO, 2014). Una de las posibilidades de continuar con esta tendencia, es implementar alta tecnología (la tecnología más avanzada disponible), la cual ha demostrado en países como India y Brasil, que puede reducir los costos de producción en más de 50%, en comparación con los costos que se alcanzan en México (Pérez Vázquez et al, 2018).

Desafortunadamente, no todas las empresas cuentan con los recursos para lograr dicha incorporación, en especial las que se encuentran en países en desarrollo. Tomando en cuenta lo anterior, como una posible opción para asistir en el aumento de la calidad y productividad de las pequeñas y medianas empresas, se propone la realización de un clasificador de productos agrícolas empleando *deep learning* (sub área de la inteligencia artificial) y visión artificial. En una primera etapa, se propone la realización de una aplicación de escritorio, la cual puede realizar tareas de clasificación y/o detección de productos en imágenes y en tiempo real, mediante el algoritmo de detección YOLO (You Only Look Once). En la etapa de pruebas los

## INTRODUCCIÓN

modelos YOLOv4, YOLOR y YOLOX, alcanzaron una gran velocidad y una precisión mayor al 91%, resultando de utilidad en el proceso de selección de producto que no cumple con los estándares que dicta la empresa, al momento de pasar por las bandas transportadoras.

### **1.2 Planteamiento del problema.**

Actualmente, existen pequeñas y medianas empresas del sector agrícola, que buscan optimizar parte de sus procesos, para mejorar su productividad. Donde la clasificación del producto en base a su calidad, se realiza de manera manual, al momento de pasar por las bandas transportadoras; aunque en las últimas décadas, las inspecciones manuales han tenido problemas para mantener la constancia y garantizar una eficacia de detección satisfactoria (Hongkun T. et al, 2020).

A pesar de la existencia de sistemas capaces de dar solución a este tipo de situaciones, usualmente se encuentran instalados en máquinas clasificadoras. Dichas alternativas resultan costosas, en especial para las empresas anteriormente mencionadas.

### **1.3 Alcances y limitaciones**

El clasificador solo corresponde a la parte del software. Se requiere una gran cantidad de imágenes para tener mejores resultados y es recomendable tener buenos recursos computacionales.

El software es una aplicación de escritorio, la cual tiene la capacidad de hacer detecciones en imágenes (al momento productos como manzana, naranja, plátano y chile jalapeño), al tener la opción de seleccionarla desde los archivos de la computadora, siendo los formatos aceptados .jpg y .jpeg. También se tiene la opción de detección en tiempo real por medio de una cámara web.

La velocidad de detección depende en gran medida de los recursos del equipo de cómputo donde se esté ejecutando la aplicación.

### **1.4 Justificación**

Uno de los retos de la agricultura en la actualidad, es mejorar la productividad; las empresas del sector agrícola están conscientes de la necesidad de innovar para lograr este cometido. Una de las maneras de promover este desarrollo es mediante los sistemas de visión computarizada, los cuales se han utilizado ampliamente en diferentes ámbitos de los segmentos del mercado de producción agrícola y alimentaria, evitando el alto costo y la baja eficiencia de las operaciones tradicionales (Hossain et al, 2019), donde se encuentra la inspección de calidad, la cual ayuda a juzgar y determinar la calidad de los productos, además de promover su comercialización (Gongal et al, 2015).

Se buscará apoyar en los procesos manuales de control de calidad de las pequeñas y medianas empresas mexicanas, para lograr una mayor eficiencia al momento de clasificar el producto agrícola de acuerdo a los criterios de calidad de las mismas. Al aumentar la calidad y productividad, se contribuye a mejorar la satisfacción del cliente, la competitividad y propiciar el aumento de la exportación.

### **1.5 Objetivo**

#### **1.5.1 Objetivo general**

Desarrollar un software capaz de clasificar productos agrícolas, mediante visión artificial.

#### **1.5.2 Objetivo específico**

- Dada la imagen de un producto, se tomará como el patrón de referencia con sus respectivos márgenes de tolerancia, para clasificarlo en la categoría correspondiente.

## INTRODUCCIÓN

- Aplicar modernos algoritmos de detección en tiempo real.
- Incluir la opción de añadir una imagen desde los archivos del usuario para realizar detecciones.
- Incluir la opción de detección en tiempo real por medio de la cámara web del usuario.
- Recopilar imágenes para formar el que se utilizará para la etapa de entrenamiento y la etapa de pruebas.

### **1.6 Hipótesis**

Una vez desarrollado el software, tendrá la capacidad de detectar productos agrícolas (fruta/verdura, con los cuales se entrenó el modelo de red neuronal) y clasificarlo en la categoría correspondiente de una manera rápida y precisa. Además, el software tendrá la capacidad de ser empleado para otro tipo de producto, sin demasiados cambios posteriores al sistema.

## II. ESTADO DEL ARTE

En la actualidad existen algunos métodos para la clasificación y/o selectividad de productos agrícolas. Dentro de los métodos se encuentra el manual, donde los trabajadores se encargan de seleccionar el producto, sin embargo, resulta menos productivo para la empresa.

Otro método consiste en utilizar máquinas clasificadoras de alimentos las cuales emplean distintas tecnologías, tales como cámaras, iluminación, software de aprendizaje, etc., para seleccionar el producto de acuerdo a las características de la empresa. Dependiendo de las especificaciones de la máquina, puede resultar muy costosa, en especial para las Pymes.

La visión artificial al proporcionar grandes beneficios a la industria alimentaria, ha sido objeto de estudio en los últimos años. En este apartado se habla acerca de algunos artículos de investigación, referentes al desarrollo de software, aplicado en tareas de control de calidad, para productos agrícolas.

Heras (2017), realizó un clasificador de imágenes (utilizando el lenguaje de programación Python) con cuatro clases de frutas; las imágenes se obtuvieron en internet y fotografías de creación propia. Utilizó la técnica de extracción del histograma a color en tres dimensiones, y la aplicación de la máquina de aprendizaje supervisado *Random Forest* en la clasificación. La máquina de aprendizaje clasifica las imágenes según el color más cercano al que se le entrenó, lo cual resulta en una limitación al momento de intentar clasificar dos imágenes de frutas diferentes, con colores muy cercanos entre sí.

Sucari León et al (2020), comprobaron la efectividad de aplicar la visión artificial en los agronegocios. Haciendo uso del clasificador bayesiano implementado en Octave, realizaron un sistema para reconocer los patrones de 6 variedades de frutas, considerando cuatro características (tamaño, forma, color y textura).

Guillazca G. et al (2020), presentaron un prototipo capaz de supervisar, identificar y clasificar productos agrícolas, utilizaron bananas, naranjas, plátano verde y manzanas. Además, el prototipo envía los resultados a su aplicación web. Emplearon redes neuronales convolucionales (CNN) entrenadas a partir del algoritmo de aprendizaje supervisado *K-Nearest Neighbor* (KNN).

Aguilar-Alvarado y Campoverde-Molina (2020), entrenaron un modelo de visión artificial utilizando CNN, para la clasificación de trece categorías de frutas, logrando obtener un 87% de eficiencia.

Chung y Van Tai (2019), examinaron la utilización de *deep learning* para el reconocimiento de frutas. Propusieron un modelo basado en el algoritmo EfficientNet, obteniendo una precisión del 95.67 %.

Figueroa y Roa (2016), implementaron un sistema para la identificación del estado de maduración de granadillas, a partir del reconocimiento de imágenes. Utilizaron Python, la librería OpenCv y el método de agrupamiento K-medias. El sistema logró obtener 92,6% de probabilidad para clasificar correctamente las granadillas.

Megha y Lakshmana (2016), presentaron un método para la clasificación automática del tomate. El sistema consta de dos partes: el sistema de manipulación de la fruta y el módulo de procesamiento de imágenes. El primero, se utiliza para mover el tomate en la cinta transportadora y para la adquisición de imágenes. En el segundo, las imágenes del tomate adquirido se analizan mediante técnicas de procesamiento de imágenes para determinar si el tomate es defectuoso o no defectuoso, y si está maduro o inmaduro. El módulo de procesamiento de imágenes, se implementó utilizando MATLAB, además se empleó una red neural pre entrenada de tres capas, para la clasificación. El software clasificó la imagen del tomate como defectuosa/no defectuosa y madura/no madura con una precisión del 100% y 96,47% respectivamente. Aunque se indicó que se necesitan mejoras especialmente en la velocidad y precisión antes de implementarlo en el campo.

Jyoti Jhawar (2016), presentó un sistema para clasificar de manera automática naranjas. Dada su imagen en un solo color de  $640 \times 480$  píxeles de resolución, tomada dentro de una caja especial diseñada con 430 lux de intensidad de luz en su interior, por una cámara digital; se clasifica 4 clases diferentes en base al nivel de madurez y en 3 clases diferentes acorde al tamaño. Dentro de las técnicas utilizadas, Regresión lineal (Linear regression) mostró mejores resultados, cerca de un 98%. El autor menciona posibles alcances o mejoras que puede tener el sistema propuesto, por ejemplo, la detección en frutos dañados y otras variedades de la naranja.

El proyecto realizado por Raj R. et al (2021) en la India, comparó la precisión de tres técnicas para la detección de imágenes de tomate en los cultivos, en base al contenido morfológico y color. Con el algoritmo de detección YOLO se logró un resultado de 85%, las redes neuronales de convolución un 63% y la detección de color en Matlab un 74%.

Parico y Ahamed (2021) experimentaron con distintas variantes del modelo YOLOv4, terminando por utilizar la versión YOLOv4-512 y en conjunto con el algoritmo de seguimiento Deep SORT, se realizó un contador de peras en los cultivos en tiempo real con una media de precisión de alrededor del 96%, el cual puede ser utilizado en aplicaciones móviles con el fin de apoyar a los agricultores de Japón en la gestión logística de las peras.

Los resultados de los estudios antes mencionados demuestran un panorama positivo y concuerdan con la idea principal del estudio realizado por Hongkun T. et al (2020), en el cual menciona que la combinación de la tecnología de visión computarizada y algoritmos de inteligencia artificial, mejoran el rendimiento de procesos agrícolas.

### III. MARCO TEÓRICO

#### 3.1 Lenguaje de programación Python [20]

Python es lo que se conoce como un lenguaje de programación de alto nivel, de propósito general, dinámico e interpretado. Interpretado significa que Python no se compila directamente en lenguaje de máquina antes de ser ejecutado por el ordenador. En su lugar, es leído y ejecutado por otro programa o un intérprete, que luego traduce el código a lenguaje de máquina que el procesador del ordenador puede entender.

Los lenguajes de programación interpretados, son generalmente más pequeños en tamaño y tienen características como la codificación dinámica, lo que significa que los tipos de datos se comprueban en tiempo de ejecución y, por tanto, el tipo de la variable puede cambiar a lo largo de su vida.

La parte de alto nivel de la composición de Python se refiere a la forma en la que utiliza elementos de lenguaje natural, por lo que, su sintaxis es fácil de leer y es de naturaleza similar al inglés. Los lenguajes de programación de alto nivel automatizan tareas como la gestión de la memoria y otros procesos informáticos de bajo nivel y, en general, tienen mucha abstracción del sistema informático.

Python se encuentra entre los lenguajes de programación de más rápido crecimiento del mundo y es utilizado por ingenieros de software, matemáticos, analistas de datos, científicos, ingenieros de redes, estudiantes y contables. El mayor punto fuerte de Python es la enorme colección de bibliotecas estándar que se pueden utilizar para lo siguiente:

- Aprendizaje automático
- Aplicaciones GUI (como Kivy, Tkinter, PyQt, etc.)
- *Frameworks* web como Django
- Procesamiento de imágenes (como OpenCV, Pillow)



- *Web scraping* (como Scrapy, BeautifulSoup, Selenium)
- Marcos de pruebas
- Multimedia
- Computación científica
- Procesamiento de textos y muchos más

### **3.2 OpenCV [18]**

Es una biblioteca de software de código abierto, utilizada para proporcionar una infraestructura común para las aplicaciones de visión por computadora y para acelerar el uso de la percepción artificial en los productos comerciales.

La biblioteca cuenta con más de 2 500 algoritmos optimizados, que incluyen un amplio conjunto de algoritmos de visión por ordenador y de aprendizaje automático, tanto clásicos como de última generación. Estos algoritmos pueden utilizarse para detectar y reconocer rostros, identificar objetos, clasificar acciones humanas en vídeos, seguir los movimientos de la cámara, seguir objetos en movimiento, extraer modelos 3D de objetos, producir nubes de puntos 3D a partir de cámaras estereoscópicas, unir imágenes para producir una imagen de alta resolución de toda una escena, encontrar imágenes similares a partir de una base de datos de imágenes, eliminar los ojos rojos de las imágenes tomadas con flash, seguir los movimientos de los ojos, reconocer paisajes y establecer marcadores para superponerlos a la realidad aumentada, etc.

### **3.3 TensorFlow [25]**

Es una plataforma integral de código abierto para el aprendizaje automático desarrollado por Google. Cuenta con un ecosistema completo y flexible de herramientas, bibliotecas y recursos de la comunidad que permite a los investigadores impulsar el estado del arte en machine learning (aprendizaje automático) y a los desarrolladores crear y desplegar fácilmente aplicaciones potenciadas por machine learning.

TensorFlow acepta datos en forma de matrices multidimensionales de dimensiones superiores llamadas tensores. Las matrices multidimensionales son muy útiles para manejar grandes cantidades de datos. Funciona sobre la base de grafos de flujo de datos que tienen nodos y aristas. Como el mecanismo de ejecución está en forma de grafos, es mucho más fácil ejecutar el código de TensorFlow de forma distribuida en un clúster de ordenadores mientras se utilizan las unidades de procesamiento gráfico (GPUs).

### **TensorFlow ofrece una API en Python**

Esta biblioteca proporciona una API de alto nivel y no se necesita una codificación compleja para preparar una red neuronal, configurar o programar una neurona.

### **TensorFlow es compatible con dispositivos de cálculo tanto de CPU como de GPU**

Las aplicaciones de *deep learning* son muy complicadas y el proceso de entrenamiento requiere demasiados cálculos. Se requiere mucho tiempo debido al gran tamaño de los datos, e implica varios procesos iterativos, cálculos matemáticos, multiplicaciones de matrices, etc. Si se realizan estas actividades en una unidad central de procesamiento (CPU) normal, por lo general demoraría mucho más.

### **3.4 Keras [16]**

Keras es una API de *deep learning* de alto nivel desarrollada por Google. Está escrita en Python y se utiliza para facilitar la implementación de redes neuronales. Keras propiamente dicho no realiza sus propias operaciones de bajo nivel, como los productos tensoriales y las convoluciones; depende de un motor *back-end* para ello. Aunque Keras soporta múltiples motores de *back-end*, su principal (y por defecto) es TensorFlow. La API de Keras viene empaquetada en TensorFlow como `tf.keras`.

Las estructuras de datos principales de Keras son las capas y los modelos. Hay dos tipos principales de modelos disponibles en Keras:

- El modelo secuencial, el cual es una pila lineal de capas.
- La API funcional de Keras, una forma de crear modelos más flexibles. La API funcional puede manejar modelos con topología no lineal, capas compartidas e incluso múltiples entradas o salidas. La idea principal es que un modelo de aprendizaje profundo suele ser un grafo acíclico dirigido (DAG) de capas. Así que la API funcional es una forma de construir grafos de capas.

### 3.5 Deep learning [12][27][28]

Inteligencia artificial, *machine learning*, *deep learning* y redes neuronales, son términos que suelen utilizarse indistintamente. Pero en realidad, *machine learning* es un subconjunto de la inteligencia artificial. *Deep learning* es un subconjunto del *machine learning* y las redes neuronales constituyen la columna vertebral de los algoritmos de *deep learning*.

Es el número de capas de nodos (profundidad), de las redes neuronales lo que distingue a una red neuronal simple de un algoritmo de *deep learning*, el número de capas deben ser tres o más. Estas redes neuronales intentan simular el comportamiento del cerebro humano permitiéndole aprender de grandes cantidades de datos. *Deep learning* impulsa muchas aplicaciones y servicios de inteligencia artificial que mejoran la automatización, realizando tareas analíticas y físicas sin intervención humana.

*Deep learning* se distingue del *machine learning* clásico por el tipo de datos con los que trabaja y los métodos en los que aprende. Los algoritmos de *machine learning* aprovechan los datos estructurados y etiquetados para hacer predicciones, lo que significa que las características específicas se definen a partir de los datos de entrada para el modelo y se organizan en tablas. Esto no significa necesariamente que no se utilicen datos no estructurados; solo significa que,

si lo hacen, suelen pasar por algún tipo de preprocesamiento para organizarlos en un formato estructurado.

*Deep learning* elimina parte del preprocesamiento de los datos que suele implicar el *machine learning*. Estos algoritmos pueden inferir y procesar datos no estructurados, como texto e imágenes, y automatizan la extracción de características, eliminando parte de la dependencia de los expertos humanos.

Los modelos de *machine learning* y de *deep learning* también son capaces de realizar diferentes tipos de aprendizaje, que suelen clasificarse en aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo.

El aprendizaje supervisado utiliza conjuntos de datos etiquetados para categorizar o hacer predicciones; esto requiere algún tipo de intervención humana para etiquetar los datos de entrada correctamente.

Por el contrario, el aprendizaje no supervisado no requiere conjuntos de datos etiquetados y, en su lugar, detecta patrones en los datos, agrupándolos por cualquier característica distintiva. El aprendizaje por refuerzo es un proceso en el que un modelo aprende a ser más preciso a la hora de realizar una acción en un entorno basándose en la retroalimentación para maximizar la recompensa.

Las redes neuronales de *deep learning*, o redes neuronales artificiales, intentan imitar el cerebro humano mediante una combinación de entradas de datos, pesos y sesgos. Estos elementos trabajan conjuntamente para reconocer, clasificar y describir con precisión los objetos de los datos.

Las redes neuronales profundas están formadas por múltiples capas de nodos interconectados, cada una de las cuales se basa en la capa anterior para refinar y optimizar la predicción o la categorización. Esta progresión de cálculos a través de la red se denomina propagación hacia

delante. Las capas de entrada y salida de una red neuronal profunda se denominan capas visibles. La capa de entrada es donde el modelo de aprendizaje profundo recibe los datos para su procesamiento, y la capa de salida es donde se realiza la predicción o clasificación final.

Otro proceso llamado retropropagación, utiliza algoritmos como el descenso de gradiente, para calcular los errores en las predicciones y luego ajusta los pesos y los sesgos de la función moviéndose hacia atrás a través de las capas en un esfuerzo por entrenar el modelo.

Juntos, la propagación hacia delante y la retropropagación permiten a una red neuronal hacer predicciones y corregir los errores. Con el tiempo, el algoritmo se vuelve gradualmente más preciso.

Lo anterior describe el tipo más simple de red neuronal profunda, sin embargo, los algoritmos de *deep learning* son increíblemente complejos, y existen diferentes tipos de redes neuronales para abordar problemas o conjuntos de datos específicos. Por ejemplo:

- Las redes neuronales convolucionales (CNN), utilizadas principalmente en aplicaciones de visión por ordenador y clasificación de imágenes, pueden detectar características y patrones dentro de una imagen, lo que permite realizar tareas como la detección o el reconocimiento de objetos.
- Las redes neuronales recurrentes (RNN) se utilizan normalmente en aplicaciones de reconocimiento del lenguaje natural y del habla, ya que aprovechan los datos secuenciales o de series temporales.

El aprendizaje profundo requiere una enorme cantidad de potencia de cálculo. Las unidades de procesamiento gráfico (GPU) de alto rendimiento son ideales porque pueden manejar un gran volumen de cálculos en múltiples núcleos con abundante memoria disponible. Sin embargo, la gestión de múltiples GPUs puede crear una gran demanda de recursos internos y ser increíblemente costosa de escalar.

### 3.6 Metodología Kanban [15]

Empezó por utilizarse en los procesos de fabricación, pero con el tiempo se ha ido empleando en otros ámbitos como lo es el desarrollo de software. La traducción de la palabra kanban del japonés al español es tablero visual o señal. El tablero básico de Kanban se compone de tres columnas (Ver Figura 3.1):

- Requested (Por hacer)
- In Progress (En proceso)
- Done (Hecho)

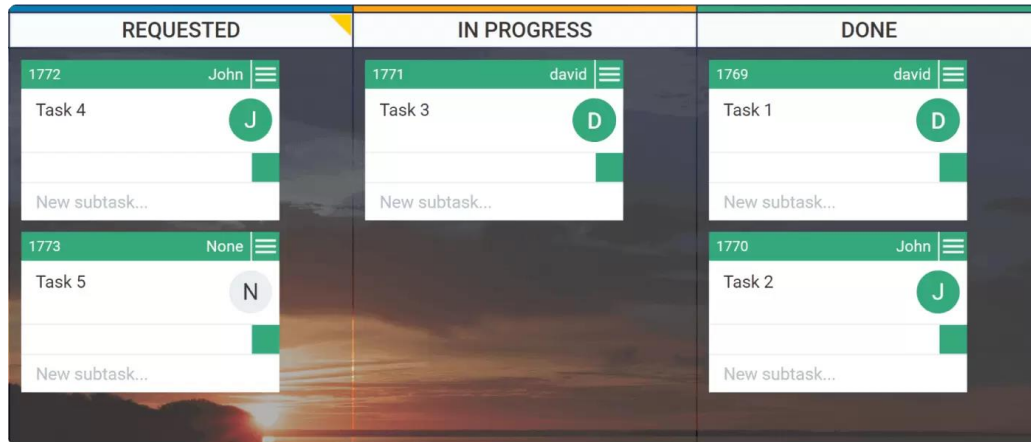


Figura 3.1 Tablero Kanban (kanbanize, 2022).

El método Kanban se formuló como un enfoque de cambio incremental y evolutivo de procesos y sistemas para las organizaciones de trabajo del conocimiento. Se centra en hacer las cosas, y sus fundamentos pueden desglosarse en dos tipos de principios y seis prácticas.

#### 3.6.1 Principios

##### De la gestión del cambio

- Empieza con lo que haces ahora

- Comprometerse a buscar e implementar cambios incrementales y evolutivos
- Animar el liderazgo en todos los niveles

### **De prestación de servicios**

- Centrarse en las necesidades y expectativas del cliente
- Gestionar el trabajo, no los trabajadores
- Revisar periódicamente la red de servicios

### **3.6.2 Prácticas**

#### **Visualizar el flujo de trabajo**

Para visualizar el proceso con un sistema Kanban, se necesita un tablero con tarjetas y columnas. Cada columna del tablero representa un paso en el flujo de trabajo. Cada tarjeta representa un elemento de trabajo. El tablero Kanban en sí representa el estado real del flujo de trabajo con todos sus riesgos y especificaciones.

#### **Limitar el trabajo en curso (work in progress, WIP)**

Una de las principales funciones de Kanban es garantizar un número manejable de elementos activos en progreso en cualquier momento. Establecer un número máximo de artículos por etapa garantiza que una tarjeta sólo se mueva a la siguiente etapa cuando haya capacidad disponible. Este tipo de limitaciones permiten descubrir rápidamente las áreas problemáticas del flujo para poder identificarlas y resolverlas.

#### **Gestionar el flujo**

La gestión del flujo consiste en gestionar el trabajo, pero no a las personas. Por flujo, se refiere al movimiento de los elementos de trabajo a través del proceso de producción a un ritmo

predecible y sostenible. En lugar de micro gestionar a las personas y tratar de mantenerlas ocupadas todo el tiempo, la atención se debe enfocar en la gestión de los procesos de trabajo y en entender cómo hacer que ese trabajo pase más rápido por el sistema. Esto significa que el sistema Kanban está creando valor más rápidamente.

### **Hacer explícitas las políticas de los procesos**

No se puede mejorar algo que no se entiende. Por eso el proceso debe estar claramente definido, publicado y socializado. Cuando se está familiarizado con el objetivo común, se puede trabajar y tomar decisiones con respecto a cambios que conllevan hacia una dirección positiva.

### **Bucles de retroalimentación**

Para los equipos y las empresas que quieren ser más ágiles, la implementación de bucles de retroalimentación es un paso obligatorio. Garantizan que las organizaciones respondan adecuadamente a los posibles cambios y permiten la transferencia de conocimientos entre las partes interesadas. Esto suele realizarse con reuniones periódicas con pocos asistentes.

### **Mejorar en colaboración**

La forma de conseguir una mejora continua y un cambio sostenible dentro de una organización es mediante la aplicación colaborativa de cambios basados en métodos, comentarios y métricas científicamente probados.

### **Beneficios**

Entre los beneficios se destacan los siguientes:

- Mayor visibilidad del flujo
- Mejora de la velocidad de entrega



- Alineación entre los objetivos y la ejecución
- Mejora de la previsibilidad
- Mejora de la gestión de las dependencias
- Aumento de la satisfacción del cliente

### **3.7 Algoritmo de detección**

Todos los detectores de objetos toman una imagen como entrada y comprimen las características a través de una red neuronal convolucional, dividiéndose en dos categorías principales.

#### **Algoritmos de detección de objetos en dos etapas**

Realizan predicciones de localización y clasificación de objetos por separado. Logran la mayor precisión de detección, pero suelen ser más lentos. Destacando los siguientes:

- RCNN y SPPNet (2014)
- Fast RCNN y Faster RCNN (2015)
- Mask R-CNN (2017)
- Pyramid Networks/FPN (2017)
- G-RCNN (2021)

#### **Algoritmos de detección de objetos en una etapa**

Realizan predicciones de localización y clasificación de objetos al mismo tiempo. Este proceso consume menos tiempo y, por lo tanto, puede utilizarse en aplicaciones en tiempo real. Dentro de los principales se encuentran:

- YOLO (2016)
- SSD (2016)

- RetinaNet (2017)
- YOLOv3 (2018)
- YOLOv4 (2020)
- YOLOR (2021)
- YOLOX (2021)

Cómo se puede observar la visión artificial es un campo que está avanzando rápidamente. Cada año aparecen alternativas que podemos utilizar en diversas tareas del día a día. Debido a los beneficios que poseen los algoritmos de detección en una etapa, se optó por elegir los modelos YOLOv4, YOLOR y YOLOX para la etapa de pruebas.

### 3.7.1 YOLO (you only look once)

Se caracteriza por su velocidad al momento de detectar objetos en una imagen o video con una gran precisión, logrando ser uno de los algoritmos de detección en tiempo real con un balance notable entre dichas características. El nombre se debe a que, en una sola ejecución del algoritmo, es capaz de predecir las clases y los cuadros delimitadores correspondientes a los objetos. Como ejemplo para demostrar de manera simple el funcionamiento de este algoritmo, se muestra la detección del producto agrícola en la Figura 3.2:



Figura 3.2. Chile jalapeño. Objeto a detectar.

## MARCO TEÓRICO

La imagen se divide en una cuadrícula de tamaño  $S \times S$  (el valor puede variar dependiendo del modelo elegido). Al pasar por la red neuronal de convolución, cada celda de la cuadrícula, se encargará de detectar un número fijo de cuadros delimitadores (ya que puede existir más de un objeto por celda) y si es que detecta un objeto, cada cuadro se compondrá de la siguiente información:

- Centro del cuadro delimitador
- Ancho
- Altura
- La clase a la que el objeto pertenece
- Probabilidad de que exista un objeto en el cuadro

Durante el proceso de detección se tendrán varios cuadros en un mismo objeto, como se aprecia en la Figura 3.3:

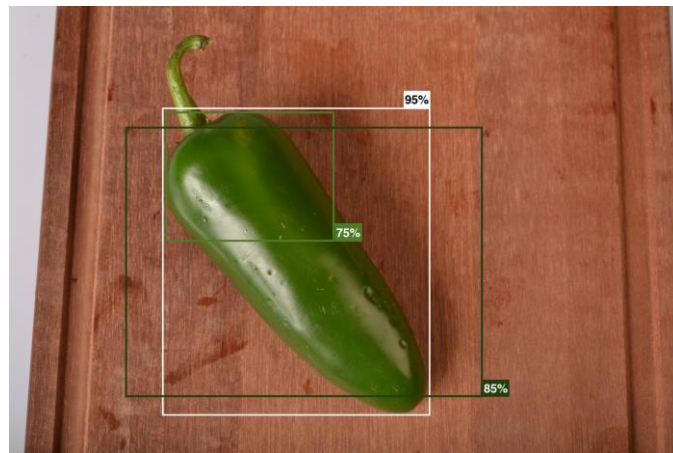


Figura 3.3. Múltiples detecciones en un mismo objeto.

Por lo que el algoritmo emplea la técnica de IoU (intersection over union) y *Non-max suppression* para concluir el proceso de detección con el mejor resultado posible. Primero se selecciona el cuadro delimitador con la mayor probabilidad, para posteriormente comparar

## MARCO TEÓRICO

dicho cuadro con los demás haciendo uso de la técnica IoU, y en base al valor que se defina como límite, se descartaran los cuadros que resulten con un IoU mayor a dicho parámetro, esto con el fin de evitar que existan cuadros detectando al mismo objeto. En el presente ejemplo quedarían solo dos cuadros, como se muestra en la Figura 3.4:

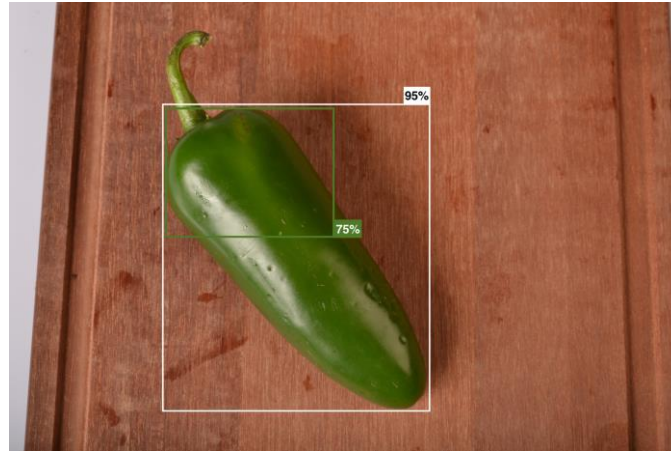


Figura 3.4. Detecciones restantes tras la aplicación de la técnica IoU.

Después se aplica la técnica de *non-max suppression*, para terminar de descartar los cuadros que tienen una menor probabilidad, como se puede apreciar en la Figura 3.5:

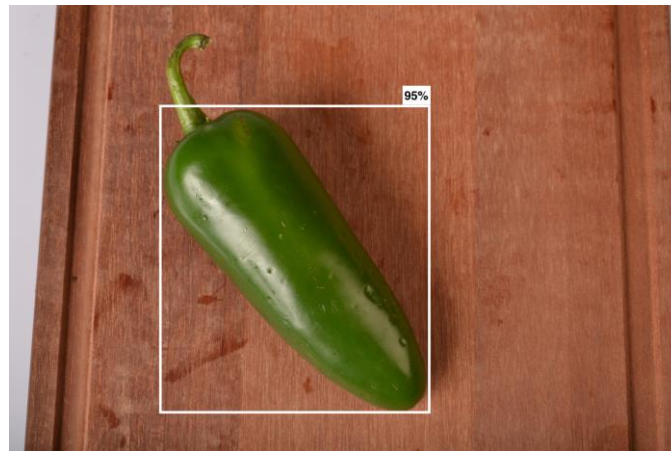


Figura 3.5. Detección restante tras la aplicación de la técnica *non-max suppression*.

Cabe mencionar que uno de los escenarios en los que el algoritmo YOLO tiene problemas para la detección, es al momento de detectar demasiados objetos que están muy cerca entre sí, o bien cuando son muy pequeños respecto al tamaño de la imagen.

### 3.7.2 YOLOv4 arquitectura [23]

En la Figura 3.6 se aprecia la arquitectura de los algoritmos de detección de una y dos etapas.

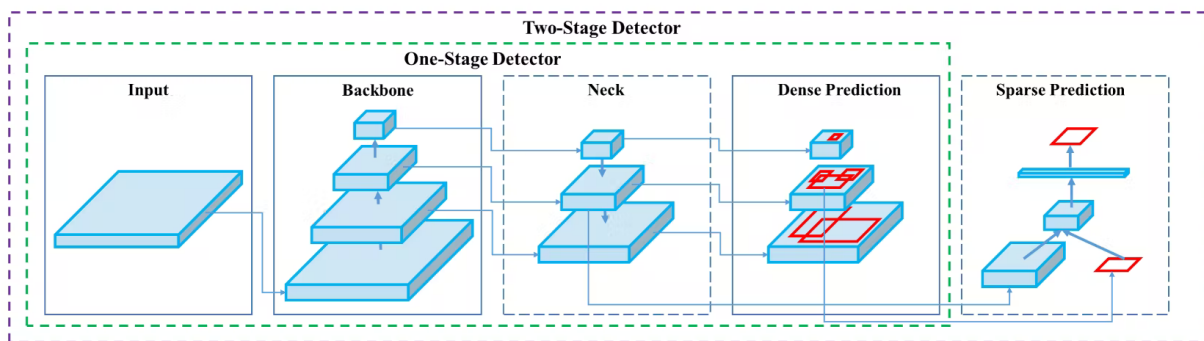


Figura 3.6. Arquitectura algoritmos de detección (Bochkovskiy et al, 2020).

#### Backbone

Es una red neuronal profunda compuesta principalmente por capas de convolución. El objetivo principal de la columna vertebral es extraer las características esenciales, la selección de la columna vertebral es un paso clave que mejorará el rendimiento de la detección de objetos. A menudo se utilizan redes neuronales pre entrenadas para entrenar la columna vertebral. El *backbone* de YOLOv4 está compuesto por la red CSPDarknet53.

#### Neck

Es la parte en la que tiene lugar la agregación de características. Recoge los mapas de características de las diferentes etapas del *backbone* y luego los mezcla y combina para prepararlos para el siguiente paso. En el presente modelo, se constituye por lo siguiente:

- SPP - Bloque adicional: Se añade un bloque adicional llamado SPP (Spatial Pyramid Pooling) entre la red troncal CSPDarkNet53 y la red de agregación de características (PANet), esto se hace para aumentar el campo receptivo y separar las características de contexto más significativas y no tiene casi ningún efecto sobre la velocidad de funcionamiento de la red.
- PANet (red de agregación de rutas): YOLOv4 utiliza una red de agregación de rutas modificada, principalmente como una mejora de diseño para hacerla más adecuada para el entrenamiento en una sola GPU.

### Head

Su función principal es localizar los cuadros delimitadores y realizar la clasificación, para esto utiliza el modelo YOLOv3. Además de esto, en YOLOv4 se introdujeron dos nuevos términos, los cuales corresponden a métodos que se utilizan en la parte del *backbone* y del detector:

- *Bag of Freebies* (BoF): Mejoran el rendimiento de la red sin aumentar el tiempo de inferencia, y la mayoría son técnicas de aumento de datos.
- *Bag of Specials* (BoS): Estas estrategias añaden incrementos marginales al tiempo de inferencia, pero aumentan significativamente el rendimiento.

### 3.8 Transfer learning

Consiste en adaptar un modelo pre entrenado para un propósito y usarlo para otro. Las características aprendidas a partir de un gran conjunto de datos deberían generalizarse a nuevas tareas.

Dicho modelo es generalmente entrenado con enormes *datasets*, los pesos obtenidos del modelo se pueden reutilizar en otras tareas (en este caso de clasificación), por lo que resulta muy

conveniente si se tiene una *dataset* pequeño. Otra de los motivos para usar esta técnica, es cuando no se tienen los recursos computacionales recomendados, logrando un entrenamiento mucho más rápido. *Transfer learning* consiste principalmente en los siguientes pasos:

- Obtener el modelo pre entrenado.
- Crear un modelo base.
- Congelar capas de la red neuronal.
- Añadir nuevas capas entrenables.
- Entrenar el modelo con nuestro *dataset*.
- Mejorar el modelo con la técnica conocida como *fine-tuning*.

### **3.9 Data augmentation**

Es una técnica utilizada para incrementar la diversidad del conjunto de entrenamiento, donde se aplican transformaciones aleatorias, por ejemplo, rotaciones, recortes o zoom.

### **3.10 Overfitting**

Es un inconveniente que puede surgir al entrenar el modelo de red neuronal, lo que ocasiona que el modelo sea incapaz de reconocer nueva información de entrada y solo es capaz de reconocer información demasiado parecida al conjunto de entrenamiento. En otras palabras, indica un aprendizaje excesivo o memorización del conjunto de entrenamiento.

### **3.11 Algoritmo de optimización**

El motivo del entrenamiento del modelo de red neuronal es minimizar la función costo, encontrando los pesos adecuados. En el actual contexto cuando se habla de optimización, se refiere al entrenamiento del modelo de forma iterativa, donde se obtiene una evaluación máxima

o mínima de la función, comparando los resultados en cada iteración cambiando los hiperparámetros en cada paso, hasta alcanzar los resultados adecuados.

Para lograr dicho cometido, se hace uso de algoritmos de optimización, entre los que destaca Adam (Adaptive Moment Estimation). El cual combina características de otros algoritmos y técnicas:

- Descenso del gradiente: algoritmo que encuentra los mínimos locales de una función diferenciable.
- Descenso Gradiente Estocástico (SGD): añade aleatoriedad al descenso del gradiente.
- Momento: acelera la convergencia.
- Gradiente adaptativo: se adapta a un ritmo de aprendizaje diferente para distintos parámetros.

### 3.12 Función de pérdida

Se utiliza para optimizar el modelo durante el entrenamiento. Por lo que el objetivo se centra en minimizar su valor. Dentro de las funciones de pérdida destaca *Cross-Entropy*. A manera de ejemplo, se parte de una tarea de clasificación hipotética, donde se tienen tres clases (Manzana, Pera, Mango) y las probabilidades de salida [0.75, 0.2, 0.05] (este formato se conoce como *one-hot encoded*) de que tal imagen pertenezca a una de ellas.

El propósito de la función *Cross-Entropy* (también llamada pérdida logarítmica), consiste en tomar esas probabilidades y medir los valores verdaderos o deseados, siendo en este caso [1, 0, 0]. El objetivo es que la salida del modelo se acerque lo más posible a la salida deseada. Durante el entrenamiento del modelo, los pesos se ajustan iterativamente con el objetivo de minimizar la función de pérdida.

La función *Categorical Cross-Entropy*, se encuentra en Keras y se utiliza para calcular la pérdida *Cross-Entropy* entre las etiquetas y las predicciones. Y se hace uso de ella cuando hay



## MARCO TEÓRICO

dos o más clases en el modelo y los valores deseados se encuentran codificados de manera *one-hot*.

## IV. DESARROLLO

### 4.1 Análisis de software

La técnica empleada para recopilar la información concerniente al software fue la entrevista, la cual se realizó al director de tesis asignado (Dr. Hernán de la Garza Gutiérrez). Conforme se fue avanzando en el proyecto, se tuvieron más reuniones con el director de tesis, donde se planteaba la aplicación de distintas tecnologías y características al proyecto.

#### 4.1.1 Requisitos funcionales y no funcionales

##### Funcionales

- Incluir la opción de añadir una imagen desde los archivos del usuario para realizar detecciones.
- Incluir la opción de detección en tiempo real por medio de la cámara web del usuario.
- Aplicar modernos algoritmos de detección en tiempo real.
- Desarrollar una interfaz gráfica fácil de utilizar.

##### No funcionales

- Las clases de productos agrícolas que en un inicio el software detectará.
- El diseño/apariencia de la interfaz gráfica.

#### 4.1.2 Diseño de la aplicación

El boceto de la primera versión de la aplicación es minimalista, como se puede observar en la Figura 4.1, donde se aprecian dos opciones para la realizar la detección.

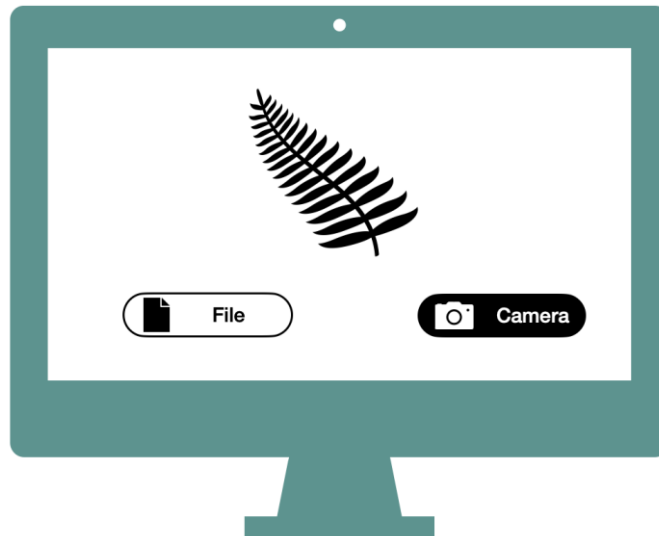


Figura 4.1. Diseño de la interfaz gráfica.

En la Figura 4.2 se describen las acciones que puede realizar el usuario:

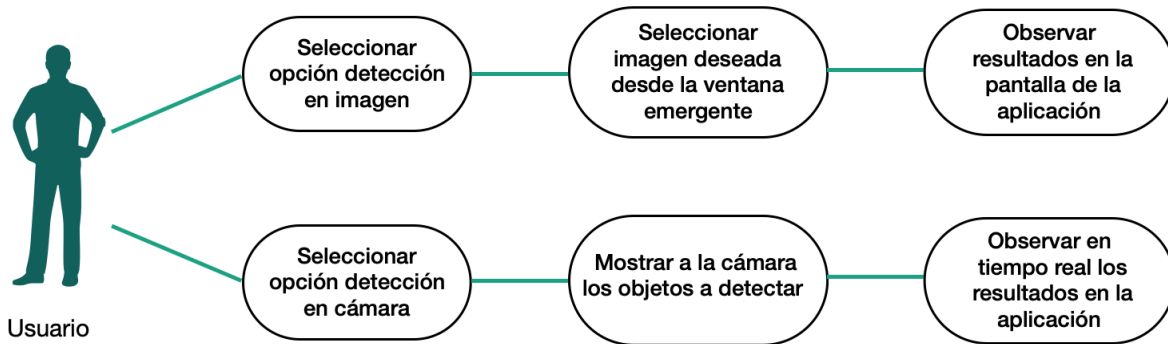


Figura 4.2. Diagrama caso de uso.

Por último, se diseñó un diagrama de componentes y distribución del software (Ver Figura 4.3), con el fin de mostrar el esquema general de la solución propuesta.

## DESARROLLO

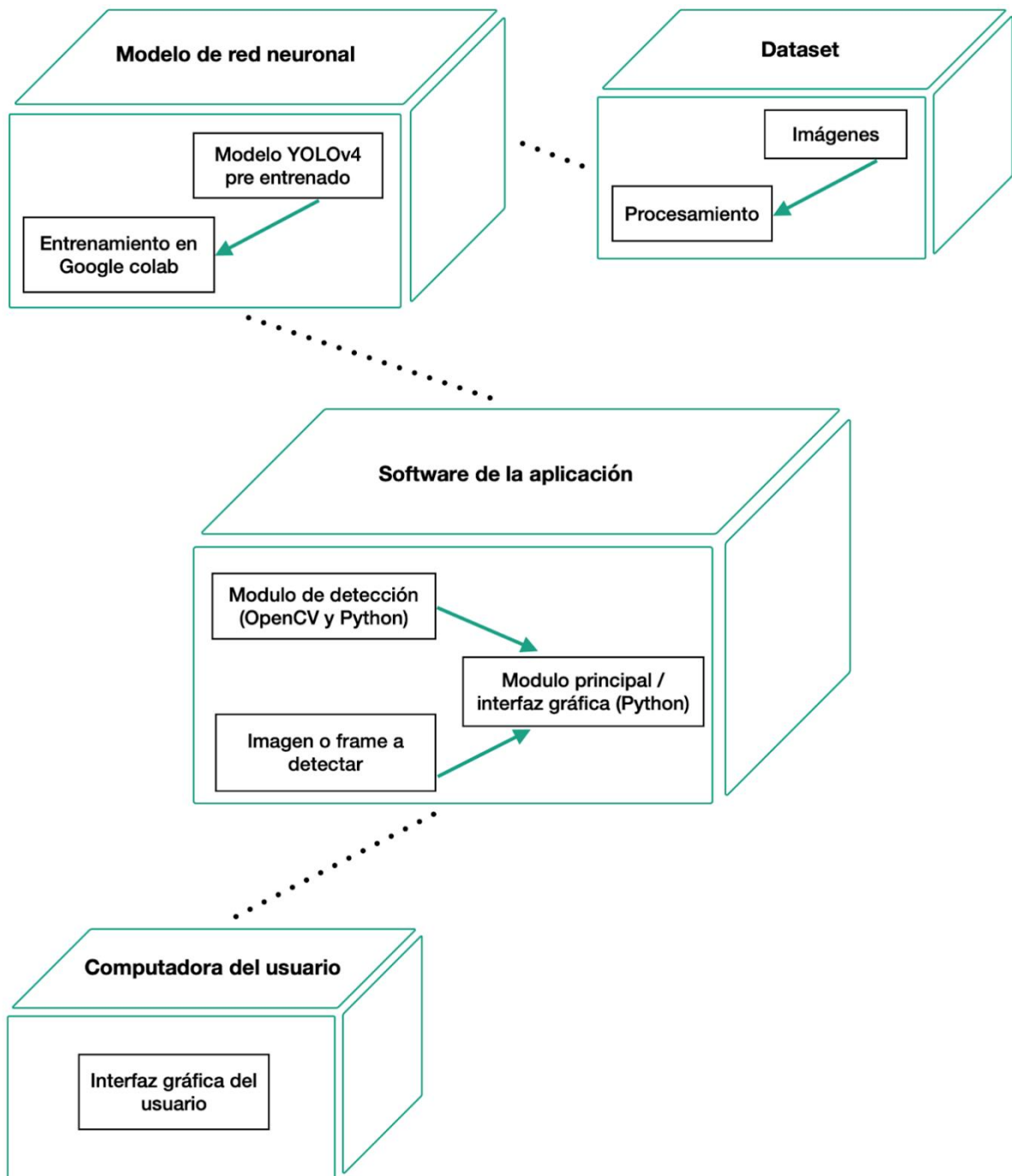


Figura 4.3. Diagrama de componentes y distribución.

## 4.2 Desarrollo de software

En este apartado se muestra el proceso correspondiente al entrenamiento de los modelos de red neuronal y el desarrollo de la aplicación. Se utilizó la metodología Kanban, la cual se menciona en la sección III. Marco teórico. El procedimiento a seguir es el siguiente:

- Creación del *dataset*: recopilar imágenes de sitios gratuitos o capturar imágenes propias.
- Preprocesamiento de imágenes: redimensionar las imágenes al tamaño que requieren los modelos de redes neuronales. En caso de contener ruido eliminarlo. Hacer uso de la técnica *data augmentation* para aumentar la cantidad de imágenes.
- Etiquetado del *dataset*: crear los cuadros delimitadores y asignar la respectiva clase a cada objeto en las imágenes.
- Entrenamiento del modelo de red neuronal: configurar el modelo acorde a las necesidades y *dataset* del proyecto.
- Evaluación del modelo de red neuronal: en caso de detectar un desempeño deficiente, se procederá a realizar configuraciones en el modelo, así como aumentar el *dataset*. En el caso contrario, se guardan los resultados deseados para la etapa de pruebas en la aplicación.
- Realizar la aplicación: crear el módulo de detección en tiempo real y en imágenes utilizando el modelo de red neuronal del procedimiento anterior. Por último, crear la interfaz gráfica incorporando los módulos de detección anteriores.

### 4.2.1 Entrenamiento modelo de red neuronal

La etapa se realizó en la plataforma web Google Colaboratory, la cual permite ejecutar código en lenguaje Python. Uno de los mayores beneficios de utilizar Colaboratory es el hecho de que desde la versión gratuita se tiene acceso a GPUs (unidades de procesamiento gráfico) y TPUs (unidad de procesamiento tensorial), componentes que resultan de suma importancia debido a su gran capacidad de realizar cálculos de manera simultánea. Se utilizó el método conocido

como *transfer learning*, el cual consiste en adaptar un modelo previamente entrenado para un propósito y usarlo para otro.

#### 4.2.1.1 Primer acercamiento: Clasificación de objetos mediante Keras

El modelo base que se utilizó fue EfficientNetB0, el cual se puede importar desde la librería Keras.

#### Obtención de la información

Potdar (2021) proporcionó un *dataset* en la página web de Kaggle, el cual contiene 14,682 imágenes repartidas entre 12 clases (6 clases corresponde a producto bueno y 6 a malo). Contiene 6 variedades de productos (manzana, plátano, naranja, tomate, melón amargo y pimiento morrón). La finalidad de haber elegido este *dataset*, es el demostrar la utilidad que tiene el *deep learning* en los procesos de calidad, en otras palabras, el reconocer la clase adecuada entre producto bueno y malo. En la Figura 4.4 y Figura 4.5 se puede observar la clase del producto pimiento morrón podrido y fresco respectivamente.



Figura 4.4. Pimiento morrón podrido (Potdar, 2021).



Figura 4.5. Pimiento morrón fresco (Potdar, 2021).

## Preprocesamiento de la información

En esta etapa se definió el conjunto de datos de entrenamiento y prueba, donde se utilizó una división del 80% y 20% respectivamente. Se asignó el tamaño por defecto del *batch* el cual es 32 y las imágenes se redimensionaron a un tamaño de 224 x 224 píxeles (Ver Figura 4.6), debido a que es el tamaño de entrada que espera el modelo elegido.

```

IMG_SIZE = (224, 224)
BATCH_SIZE = 32

train_data = image_dataset_from_directory(base_dir,
                                         label_mode="categorical",
                                         batch_size=BATCH_SIZE,
                                         image_size=IMG_SIZE,
                                         subset = "training",
                                         validation_split=0.2,
                                         seed=42)

test_data = image_dataset_from_directory(base_dir,
                                         label_mode="categorical",
                                         batch_size=BATCH_SIZE,
                                         image_size=IMG_SIZE,
                                         subset = "validation",
                                         validation_split=0.2,
                                         seed=42)

```

Figura 4.6. Preprocesamiento de la información.

## Construcción del modelo

Se definió una función *callback* llamada *ModelCheckpoint* la cual se puede apreciar en la Figura 4.7, con la finalidad de guardar los mejores pesos aprendidos por el modelo, monitoreando la precisión de la validación en cada época.

```

checkpoint_path = "model_checkpoint"
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,
                                                         save_weights_only=True,
                                                         monitor="val_accuracy",
                                                         save_best_only=True)

```

Figura 4.7. Función *ModelCheckpoint*.

Además, se creó un modelo secuencial para aplicar la técnica de *data augmentation* a las imágenes de entrada, con el fin de evitar el problema de *overfitting*; se eligieron tres modificaciones: voltear, rotar y aumentar al azar (Ver Figura 4.8).

```
data_augmentation = tf.keras.Sequential([
    preprocessing.RandomFlip("horizontal_and_vertical"),
    preprocessing.RandomRotation(0.2),
    preprocessing.RandomZoom(0.2),
], name="data_augmentation_layer")
```

Figura 4.8. Modelo secuencial.

Se creó el modelo base (EfficientNetB0) sin incluir las últimas capas, ya que se aplicó *transfer learning*. Además, para aprovechar los pesos aprendidos por el modelo base, se necesita congelar o deshabilitar la propiedad de entrenamiento del mismo (Ver Figura 4.9), de esta manera se conservan los patrones aprendidos del *dataset* ImageNet con el que fue previamente entrenado, el cual cuenta con alrededor de 14 millones de imágenes.

```
# 1. Create base model
base_model = tf.keras.applications.EfficientNetB0(include_top=False)

# 2. Freeze the base model
base_model.trainable = False
```

Figura 4.9. Modelo base.

Luego se definió la capa de entrada con sus respectivas dimensiones (Ver Figura 4.10) y se pasó al modelo *data\_augmentation* (previamente definido), después el resultado se adjuntó al modelo base.

```
# 3. Create input into the model
inputs = layers.Input(shape=IMG_SIZE + (3,), name="input_layer")

# 4. Add in data augmentation Sequential model as a layer
x = data_augmentation(inputs)

# 5. Pass the inputs to the base model after data augmentation
x = base_model(x, training=False) # put the base model in inference mode
```

Figura 4.10. Creación capa de entrada.



Se añadió una capa densa con 512 unidades o neuronas, seguida de una capa *dropout*, la última con el fin de reducir el problema del *overfitting* (Ver Figura 4.11).

```
# 6. Add a fully connected layer with 512 hidden units and ReLU activation
x = layers.Dense(512, activation='relu')(x)

# 7. Add a dropout rate of 0.5
x = layers.Dropout(0.5)(x)
```

Figura 4.11. Capa densa y *dropout*.

Después, se agregó una capa *AveragePooling*, seguida de la capa de salida con 12 unidades (correspondiente al total de las clases). Por último, se creó el modelo pasando por argumento las capas de entrada y de salida previamente configuradas como se observa en la Figura 4.12.

```
# 8. Avg pool the outputs of the base model
x = layers.GlobalAveragePooling2D(name="global_avg_pooling_layer")(x)

# 9. Create the output activation layer
outputs = layers.Dense(12, activation="softmax", name="output_layer")(x)

# 10. Combine the inputs and outputs into a model
model = tf.keras.Model(inputs, outputs)
```

Figura 4.12. Capa *AveragePooling* y de salida. Modelo final.

## Compilación y entrenamiento del modelo

Se compiló el modelo utilizando una tasa de aprendizaje de 0.005 y como optimizador *Adam*, que como se verá más adelante demostró un buen resultado. Debido a la naturaleza del proyecto se eligió la función de pérdida *categorical cross entropy* (Ver Figura 4.13).

```
model.compile(loss="categorical_crossentropy",
              optimizer=tf.keras.optimizers.Adam(0.005),
              metrics=["accuracy"])
```

Figura 4.13. Compilación del modelo.

La siguiente etapa corresponde al entrenamiento, el cual se realizó por 5 épocas. Para agilizar este proceso se redujeron los lapsos de validación a un 15% como se observa en la Figura 4.14. Además, en esta etapa se pasó por argumento la función *checkpoint\_callback* previamente definida.

```
history = model.fit(train_data,  
                    epochs=5,  
                    validation_data=test_data,  
                    validation_steps=int(0.15 * len(test_data)),  
                    callbacks=[checkpoint_callback])
```

Figura 4.14. Entrenamiento del modelo.

### Evaluación del modelo

Como en el paso anterior la validación se realizó en el 15% del *dataset* de prueba, una vez entrenado el modelo se evalúa con el 100% (Ver Figura 4.15).

```
feature_extraction_results = model.evaluate(test_data)  
feature_extraction_results
```

Figura 4.75. Evaluación del modelo.

### Resultados parciales

Se obtuvo un rendimiento del 98.8% en el *dataset* de entrenamiento y un 98.3% en el de prueba.

### Aplicación de la técnica *fine-tuning*

Para mejorar un poco más el rendimiento del modelo, se utilizó la técnica *fine-tuning*. Para empezar, se descongelaron algunas capas del modelo base (EfficientNetB0), cambiando la propiedad para permitir su entrenamiento, en este caso fueron las últimas 10 capas (Ver Figura 4.16).

## DESARROLLO

```
# Unfreeze the layers in the base model
base_model.trainable = True

# Freeze every layer except the last 10
for layer in base_model.layers[:-10]:
    layer.trainable = False
```

Figura 4.16. Utilización de la técnica *fine-tuning*.

Se volvió a compilar el modelo, pero con una menor tasa de aprendizaje (Ver Figura 4.17).

```
model.compile(loss="categorical_crossentropy",
              optimizer=tf.keras.optimizers.Adam(learning_rate=0.0005),
              metrics=["accuracy"])
```

Figura 4.17. Compilación del modelo, etapa *fine-tuning*.

Se procedió a entrenar el modelo por otras 5 épocas, de igual manera validando con un 15% de la información para agilizar el proceso (Ver Figura 4.18).

```
history_2 = model.fit(train_data,
                     epochs=10,
                     validation_data=test_data,
                     validation_steps=int(0.15 * len(test_data)),
                     initial_epoch=history.epoch[-1],
                     callbacks=[checkpoint_callback])
```

Figura 4.18. Entrenamiento del modelo, etapa *fine-tuning*.

Una vez entrenado el modelo se evaluó con el 100% de la información (Ver Figura 4.19).

```
fine_tuning_results = model.evaluate(test_data)
fine_tuning_results
```

Figura 4.19. Evaluación del modelo, etapa *fine-tuning*.

## Resultados finales

Tras la aplicación de la técnica *fine-tuning*, el modelo logró un rendimiento de un 98.48% en el *dataset* de entrenamiento y un 99.22% en el *dataset* de prueba. Los resultados parciales y finales se pueden observar en la Figura 4.20, separados por la línea verde:



Figura 4.20. Resultados del modelo. Librería Keras.

## Predicción

Se procedió a poner a prueba el modelo realizando una predicción o clasificación, la cual nos muestra la imagen original con su respectiva clase, como se aprecia en la Figura 4.21:

## DESARROLLO

Prediction: fresh\_capsicum



Figura 4.81. Predicción acertada del modelo. Librería Keras.

### 4.2.1.2 Segundo acercamiento: Algoritmo YOLO

#### Recopilación de la información

Se utilizaron imágenes de un *dataset* gratuito obtenido de la plataforma Kaggle, las cuales corresponden a tres tipos de frutas: manzana, plátano y naranja, mismas que ya están etiquetadas utilizando un formato XML. En dicho formato se incluye el tamaño de la imagen y, por cada objeto a identificar, la posición del cuadro delimitador expresando las coordenadas de la esquina superior izquierda y la esquina inferior derecha, de acuerdo a los píxeles de la imagen.

Además, se recolectaron de otras plataformas imágenes de otro producto, chile jalapeño. Entre los cuatro productos agrícolas conforman un total de 339 imágenes de distintas dimensiones. Se tomaron 262 imágenes para el conjunto de entrenamiento (80%) y el 20% restante para el conjunto de validación.

### **Data augmentation (Aumento de datos)**

Se utilizó la aplicación web Roboflow para etiquetar las imágenes correspondientes a la categoría jalapeño, así como para la modificación del etiquetado de algunas imágenes de las otras categorías.

Después del etiquetado, se realizó la etapa de preprocesamiento, en la cual se redimensionaron las imágenes a un tamaño de 416 x 416 píxeles, debido a que las imágenes deben tener las mismas dimensiones que requiere el modelo de red neuronal de convolución utilizado y aunque esta tarea se puede realizar durante el entrenamiento de la red, desarrollarlo con antelación logra una rapidez un poco mayor. También se utilizó la aplicación Roboflow para el aumento de imágenes de entrenamiento a partir de las existentes, donde se eligieron tres modificaciones:

- Girar (horizontal y vertical)
- Rotación de 90° (a favor y en contra de las manecillas del reloj)
- Exposición a la luz (entre una escala del -10% y 10%)

Estas modificaciones se aplican de forma aleatoria, por ejemplo, una imagen puede terminar girada verticalmente y con una exposición a la luz de un -5%, en cambio, otra imagen sólo puede terminar con una rotación de 90° en contra de las manecillas del reloj.

Dentro de los beneficios de esta etapa se encuentra el hecho de que se obtiene una mayor variedad y cantidad de imágenes (resultando en este caso en 806 imágenes), lo cual trae consigo un mejor desempeño en la etapa de entrenamiento.

### **Entrenamiento YOLOv4**

Lo primero fue clonar el repositorio conocido como darknet del autor del modelo (Ver Figura 4.22).

```
!git clone https://github.com/AlexeyAB/darknet
```

Figura 4.22. Repositorio darknet.

Se accedió al directorio utilizando la instrucción: ‘%cd darknet’. Luego se construyó el programa ejecutable a partir del código fuente con el comando: ‘!make’.

Se realizó la conexión con Google Drive (Ver Figura 4.23), donde se encuentra el *dataset* previamente definido y otros archivos necesarios para el entrenamiento, los cuales se mencionará su contenido en los siguientes párrafos.

```
%cd ..  
from google.colab import drive  
drive.mount('/content/gdrive')
```

Figura 4.23. Establecer conexión con Google Drive.

### Anotaciones del *dataset*

Por cada imagen debe existir un archivo txt, con el mismo nombre. Y cada txt debe contener la siguiente información por cada objeto o producto en la imagen:

- La clase expresada con un número acorde al orden definido en el archivo con terminación *.names*.
- Los siguientes atributos correspondientes al cuadro delimitador (usualmente conocido como bounding box) deben estar normalizados (su valor debe encontrarse entre 0 y 1), por lo tanto, se realizan las siguientes operaciones:
  - Centro en X = Centro en X / ancho de la imagen
  - Centro en Y = Centro en Y / altura de la imagen
  - Ancho del objeto = Ancho del objeto / ancho de la imagen
  - Altura del objeto = Altura del objeto / altura de la imagen

### Archivo .names

En este documento se expresa el nombre de las clases encontradas en el *dataset*, una en cada línea como se observa en la Figura 4.24.

```
apple|
banana
jalapeno
orange
```

Figura 4.24. Contenido del archivo .names.

### Archivos test.txt y train.txt

En estos documentos se debe especificar la ruta donde se encuentran las imágenes correspondientes al conjunto de entrenamiento y al de pruebas, un ejemplo del contenido se observa en la Figura 4.25.

```
data/dataset/train/apple_11_jpg.rf.2b18fe2b10049a5a53fd130301ce29c3.jpg
data/dataset/train/apple_11_jpg.rf.6a736988aa88ebf2172757f23c73c710.jpg
data/dataset/train/apple_11_jpg.rf.9afdf6e4dd17ce4254b11b58dc9a0dd9.jpg
data/dataset/train/apple_12_jpg.rf.449e484e824bf328178f1f4fe6a75954.jpg
data/dataset/train/apple_12_jpg.rf.a72d65be2f0740e4b804834d5e80109b.jpg
data/dataset/train/apple_12_jpg.rf.e6b21986bcf8582ce14b89eab3d53591.jpg
```

Figura 4.25. Fragmento del contenido del archivo train.txt. Donde se indican las rutas de las imágenes.

### Archivo .data

En este archivo se especifica el total de clases, la ruta de los tres documentos previamente mencionados y la ubicación donde se guardarán cada cierto número de épocas los pesos durante el entrenamiento (Ver Figura 4.26), de esta manera se puede reanudar el proceso en caso de que surjan problemas.



```
classes = 4|
train = data/train.txt
valid = data/test.txt
names = data/obj.names
backup = /mydrive/yolo_practice_2/backup
```

Figura 4.26. Contenido del archivo .data.

### Archivo .cfg (configuración)

En este archivo se encuentra la estructura del modelo YOLOv4, así como hiperparámetros y variables. Lo primero es asignar las dimensiones adecuadas acorde al conjunto de datos, como se puede apreciar en la Figura 4.27.

```
width=416
height=416
channels=3
```

Figura 4.27. Archivo de configuración. Dimensiones.

Para obtener el valor *max\_batches* (Ver Figura 4.28), se debe multiplicar el número de clases por la cantidad de 2000, pero cuando se tienen menos de 4 clases la cantidad debe ser igual a 6000. Con el valor obtenido se calcula las cantidades correspondientes al 80% y 90% que son asignadas a la variable *steps* (Ver Figura 4.29).

```
max_batches =8000
```

Figura 4.28. Archivo de configuración. Variable *max\_batches*.

```
steps=6400,7200
```

Figura 4.29. Archivo de configuración. Variable *steps*.

Lo siguiente es asignar el número de clases adecuado en las capas YOLO del modelo (Ver Figura 4.30).

```
[yolo]
mask = 6,7,8
anchors = 12, 16, 19, 36, 40, 28, 36, 75,
classes=4
```

Figura 4.30. Archivo de configuración. Capa yolo.

Por último, antes de cada una de las capas YOLO hay una capa de convolución, donde se debe cambiar la cantidad de filtros (Ver Figura 4.31) obtenidos de la siguiente manera: (Número de clases + 5) \* 3.

```
[convolutional]
size=1
stride=1
pad=1
filters=27
activation=linear
```

Figura 4.31. Archivo de configuración. Capa de convolución.

### Iniciar entrenamiento

Lo siguiente es descargar los pesos pre-entrenados para hacer uso del método *transfer learning* e iniciar el entrenamiento como se aprecia en la Figura 4.32, donde se indica la ubicación de los archivos .data y .cfg.

```
!./darknet detector train data/my_dataset.data cfg/yolov4.cfg
```

Figura 4.32. Entrenamiento del modelo YOLOv4.

## DESARROLLO

Después de completarse o al detener el entrenamiento, se tiene la opción de observar los resultados durante las iteraciones como se puede apreciar en la Figura 4.33, donde se aprecia la media de precisión indicada con color rojo y la pérdida en azul.

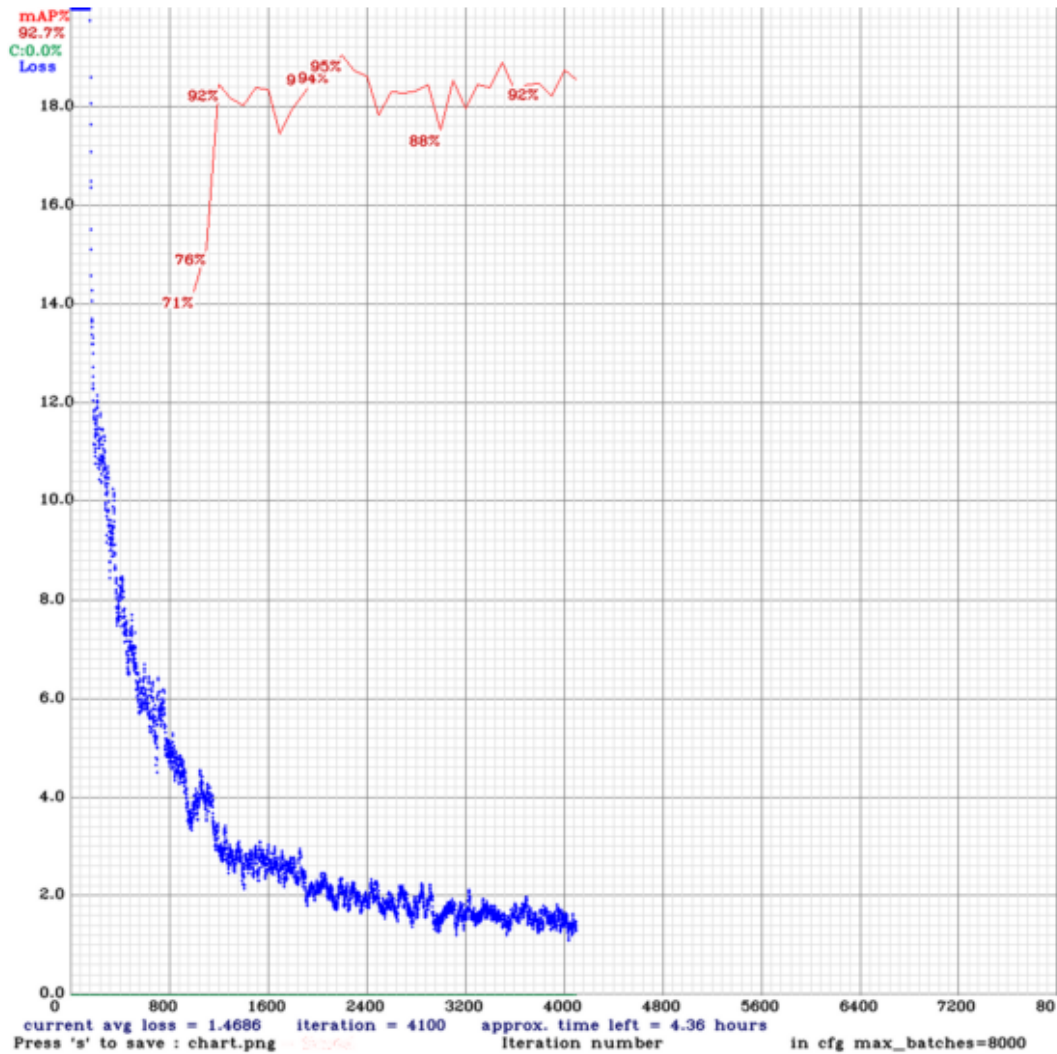


Figura 4.33. Resultados de las iteraciones del entrenamiento.

Al poseer la característica de guardar automáticamente los pesos cada cierto número de iteraciones, se puede detener el entrenamiento y retomararlo con los últimos almacenados. Así mismo, el respaldo de los pesos otorga la opción de elegir los mejores.

## Entrenamiento YOLOX y YOLOR

A grandes rasgos el proceso es similar al mostrado en YOLOv4, solo que en esta ocasión para acelerar el proceso se siguieron las guías propuestas por Roboflow, las cuales a su vez se basan en los repositorios de los autores originales de YOLOX y YOLOR. Los pasos son los siguientes:

- Instalar las dependencias y/o librerías necesarias.
- Con ayuda de la aplicación RoboFlow, se exporta el *dataset* y las anotaciones en el formato que requieren los modelos. Para YOLOX el formato es xml y para YOLOR es txt. Donde se incluye por cada objeto en su respectiva imagen, la clase perteneciente y la posición del cuadro delimitador.
- Lo siguiente es descargar los pesos pre entrenados, para a partir de ellos realizar el entrenamiento.
- Por último, se realiza el entrenamiento.

### 4.2.2 Módulo detección en tiempo real

De los modelos entrenados se eligió YOLOv4, debido a la compatibilidad que posee con OpenCV y el hecho de haber mostrado resultados favorables, los cuales se pueden observar en la siguiente sección. Los pasos necesarios para la implementación del módulo de detección en objetos en tiempo real fueron los siguientes:

#### Importar librerías

Las librerías necesarias se aprecian en la Figura 4.34, donde time se utilizó como métrica del desempeño del programa.

```
import numpy as np
import cv2
import time
```

Figura 4.34. Librerías para el desarrollo del módulo de detección.

### Definir entrada de video

Mediante la librería OpenCV se creó un objeto de la clase *VideoCapture* y por argumento se pasó el índice de la cámara, el cual para una cámara incorporada suele ser el 0 (Ver Figura 4.35).

```
camera = cv2.VideoCapture(my_camera)
```

Figura 4.35. Creación de un objeto *VideoCapture*.

### Cargar la red YOLOv4

Primero se obtuvieron los nombres de las clases del archivo `.names` (Ver Figura 4.36), especificado en la etapa de entrenamiento.

```
with open('yolo-data/obj.names') as f:
    labels = [line.strip() for line in f]
```

Figura 4.36. Procesamiento del archivo `.names`.

Se creó la red neuronal con ayuda de OpenCV, pasando por argumento los pesos y el archivo `.cfg`, del modelo previamente entrenado (Ver Figura 4.37).

```
network = cv2.dnn.readNetFromDarknet('yolo-data/yolov4.cfg',
                                     'yolo-data/yolov4.weights')
```

Figura 4.37. Creación de la red neuronal mediante OpenCV.

Se obtuvo el nombre de las capas que componen a la red (Ver Figura 4.38).

```
layers_names_all = network.getLayerNames()
```

Figura 4.38. Nombre de las capas de la red neuronal.

De las capas anteriores, se obtuvo solo las de salida con ayuda del método `getUnconnectedOutLayers()` de la librería OpenCV (Ver Figura 4.39).

```
layers_names_output = \
    [layers_names_all[layer[0] - 1] for layer in network.getUnconnectedOutLayers()]
```

Figura 4.39. Obtener capas de salida de la red neuronal.

Después se definieron dos probabilidades, las cuales dependen de las características del proyecto (Ver Figura 4.40 y Figura 4.41).

```
score_threshold = 0.5
```

Figura 4.40. Probabilidad mínima para eliminar predicciones débiles.

```
nms_threshold = 0.3
```

Figura 4.41. Umbral utilizado en la técnica de non-max suppression.

### Leer los frames de la entrada de video

Se definió una variable para capturar cada *frame* mediante el objeto *camera*, en la primera ocasión se obtienen las dimensiones del mismo (Ver Figura 4.42).

```
while True:
    _, frame = camera.read()

    if width is None or height is None:
        height, width = frame.shape[:2]
```

Figura 4.42. Capturar *frames* de la cámara.

### Obtener blob del frame

El método `blobFromImage()` retorna un *blob* de 4 dimensiones (número de *frames*, número de canales, ancho y alto), el cual es la imagen de entrada después de la sustracción de la media, la normalización y el intercambio de canales RB (red y blue respectivamente). Dicho método se puede observar en la Figura 4.43.

```
blob = cv2.dnn.blobFromImage(frame, 1 / 255.0, (416, 416),
                             swapRB=True, crop=False)
```

Figura 4.43. Obtener *blob* del *frame*.

### Detección de objetos

El objeto anterior (*blob*) se pasó como entrada a la red y posteriormente se realizó el *forward pass* a través de las capas de salida (Ver Figura 4.44).

```
network.setInput(blob)
start = time.time()
output_from_network = network.forward(layers_names_output)
end = time.time()
```

Figura 4.44. Instrucciones para la detección de objetos.

### Obtener los cuadros delimitadores

Por medio de un bucle se recorrieron las capas de salida y mediante otro se iteró a través de las detecciones. En seguida se obtuvo la clase con la probabilidad más alta a la que el objeto detectado pertenece (Ver Figura 4.45).

```

probabilities = detected_objects[5:]

current_class = np.argmax(probabilities)

current_confidence = probabilities[current_class]

```

Figura 4.45. Obtener probabilidad más alta del objeto detectado.

Se verificó si dicha probabilidad es mayor a la mínima probabilidad que se definió con antelación, de ser así, se escalan las coordenadas del cuadro delimitador al tamaño inicial del frame o imagen (Ver Figura 4.46):

```

current_box = detected_objects[0:4] * np.array([width, height, width, height])

```

Figura 4.46. Creación del cuadro delimitador.

Del anterior cuadro delimitador se obtuvieron los valores almacenados como se muestra en la primera línea de código de la Figura 4.47, y con ellos se obtuvieron las coordenadas de la esquina superior izquierda, necesarias para dibujar el cuadro en el frame o imagen.

```

x_center, y_center, box_width, box_height = current_box
x_min = int(x_center - (box_width / 2))
y_min = int(y_center - (box_height / 2))

```

Figura 4.47. Obtener coordenadas del cuadro delimitador.

### **Non-max suppression**

Se hizo uso de esta técnica para excluir algunos de los cuadros delimitadores, cuando sus probabilidades correspondientes son bajas o existe otro cuadro delimitador para esta región con una mayor probabilidad (Ver Figura 4.48).



```
results = cv2.dnn.NMSBoxes(bounding_boxes, confidences,
                           score_threshold, nms_threshold)
```

Figura 4.48. Aplicación de la técnica *Non-max suppression*.

### Dibujar cuadros delimitadores y etiquetas de clase

Se procedió a dibujar el cuadro delimitador, pasando los siguientes argumentos al método *rectangle()* de la librería OpenCV (Ver Figura 4.49):

- *Frame*
- Coordenadas del punto inicial
- Coordenadas del punto final
- Grosor del cuadro a dibujar

```
cv2.rectangle(frame, (x_min, y_min),
              (x_min + box_width, y_min + box_height),
              color_current_box, 2)
```

Figura 4.49. Dibujar cuadro delimitador.

Por último, se colocó el nombre de la clase y su respectiva probabilidad junto al cuadro delimitador (Ver Figura 4.50).

```
cv2.putText(frame, text_class_prob, (x_min, y_min - 5),
            cv2.FONT_HERSHEY_SIMPLEX, 0.5, color_current_box, 2)
```

Figura 4.50. Asignar texto junto cuadro delimitador.

### Mostrar las detecciones

Se creó una ventana OpenCV y se mostró el *frame* con las detecciones (Ver Figura 4.51).

```
cv2.namedWindow('YOLOv4 Real Time', cv2.WINDOW_NORMAL)  
  
cv2.imshow('YOLOv4 Real Time', frame)
```

Figura 4.51. Mostrar las detecciones en una ventana.

### 4.2.3 Módulo detección en imagen

El proceso es similar al anterior, correspondiendo los siguientes pasos:

- Importar librerías
- Leer imagen: Proporcionar la ruta del archivo al método *imread()* de OpenCV (Ver Figura 4.52).

```
user_img = cv2.imread(path)
```

Figura 4.52. Instrucción para leer la imagen a detectar.

- Cargar la red YOLOv4
- Obtener *blob* de la imagen
- Detección de objetos
- Obtención de cuadro delimitadores
- *Non max suppression*
- Dibujar cuadros delimitadores y etiquetas de clase
- Guardar resultados: Se puede guardar la imagen con las detecciones realizadas (Ver Figura 4.53).

```
cv2.imwrite('result.jpg', user_img)
```

Figura 4.53. Instrucción para guardar la detección.

## V. RESULTADOS

### 5.1 Resultados entorno Colab

Se evaluó cada modelo para elegir el más adecuado acorde al proyecto. Se comparó la media de precisión (mAP@0.50) y se realizó la detección en una imagen de prueba utilizando los recursos computacionales de Colab (debido a que es la versión gratuita suele asignar una GPU Tesla K80), para tener una idea de la velocidad de cada modelo bajo un mismo escenario. Las detecciones se observan en las figuras 5.1, 5.2 y 5.3, donde por cada objeto se muestra la clase con su respectiva probabilidad de pertenencia.

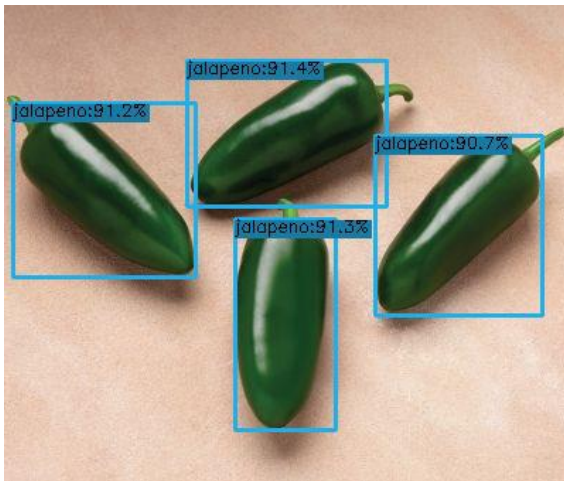


Figura 5.1. Detecciones con modelo YOLOX.

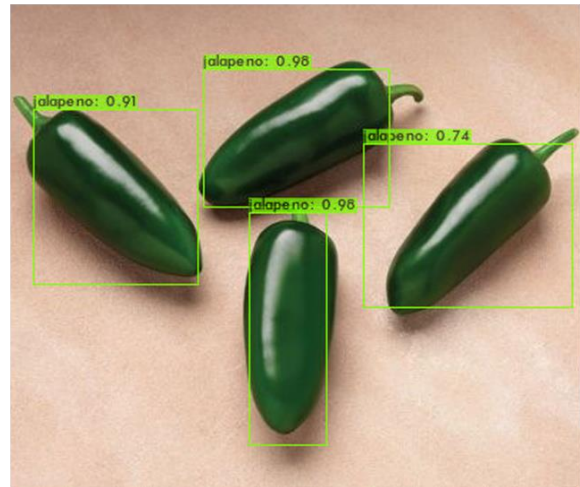


Figura 5.2. Detecciones con modelo YOLOv4.

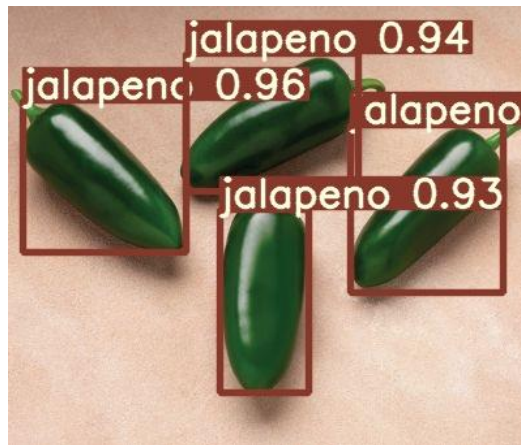


Figura 5.3. Detecciones con modelo YOLOR.

## RESULTADOS

Como se puede observar en la Figura 5.4, los resultados son favorables. Los tres modelos lograron una media de precisión por arriba del 91% y son relativamente rápidos. Estas dos características resultan atractivas para auxiliar en los procesos de control de calidad en el área de la agricultura. Por ser el modelo que mejor desempeño obtuvo y por poseer una gran integración con la librería OpenCV, se terminó por elegir YOLOv4.

Modelo	Media de precisión (mAP@0.50)	Tiempo de detección (s)
YOLOv4	95%	0.101
YOLOR	93%	0.163
YOLOX	92%	0.121

Figura 5.4. Resultados de los modelos.

### 5.2 Detecciones en la aplicación

Se utilizó la librería tkinter para el desarrollo de la interfaz gráfica, la cual se puede apreciar en la Figura 5.5, tiene dos botones los cuales corresponden a la detección en tiempo real y en imagen:

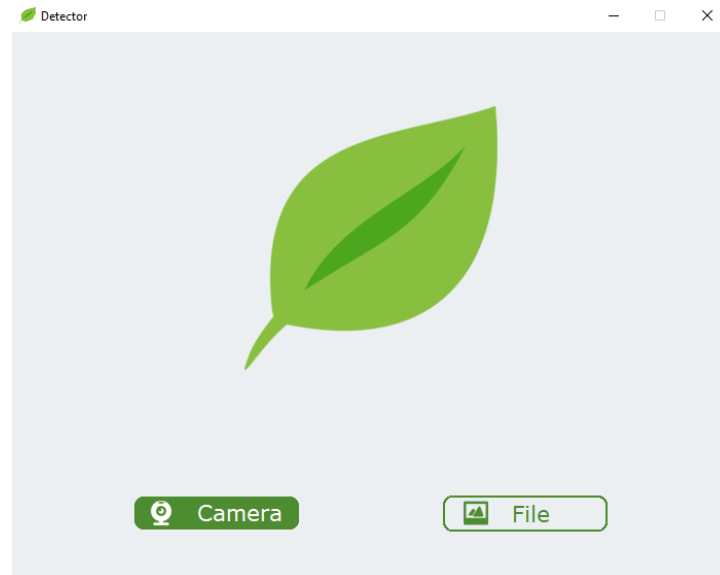


Figura 5.5. Interfaz gráfica.

## RESULTADOS

La Figura 5.6 corresponde a la detección en una imagen (opción *File* de la aplicación), donde se muestra en una nueva ventana el resultado.

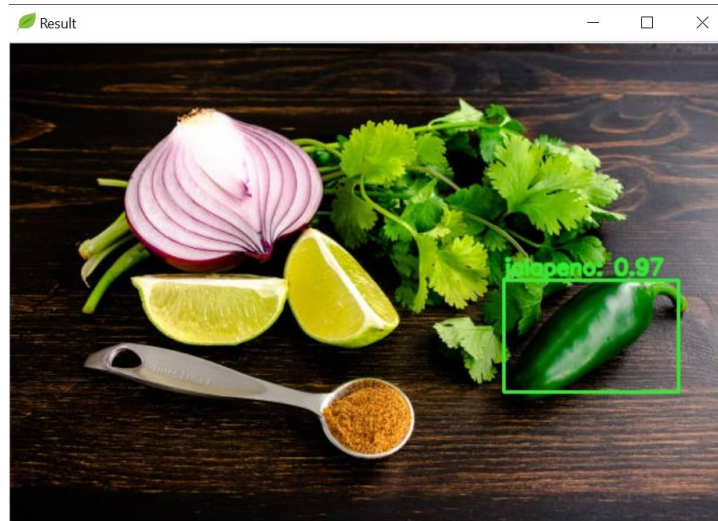


Figura 5.6. Detección en imagen.

Otras detecciones realizadas se aprecian en la Figura 5.7 y Figura 5.8, donde por cada objeto se observa el cuadro delimitador, el nombre de la clase y la probabilidad de que el objeto pertenezca a dicha clase.

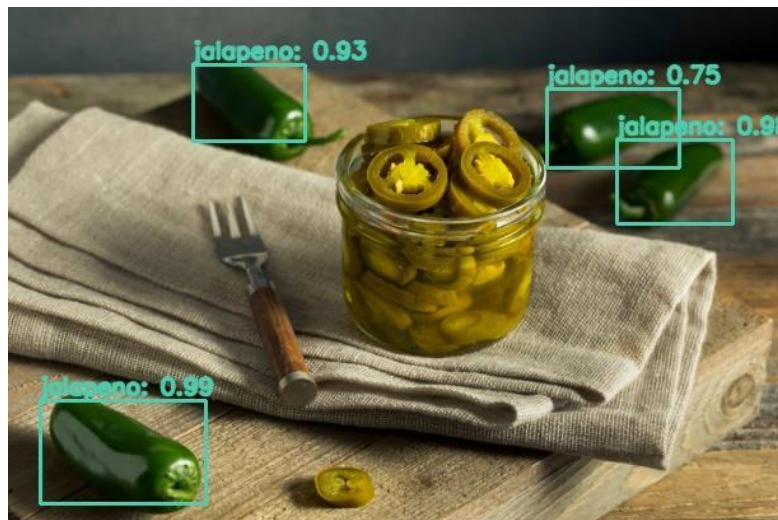


Figura 5.7. Detección en imagen. Varios objetos, misma clase.

## RESULTADOS

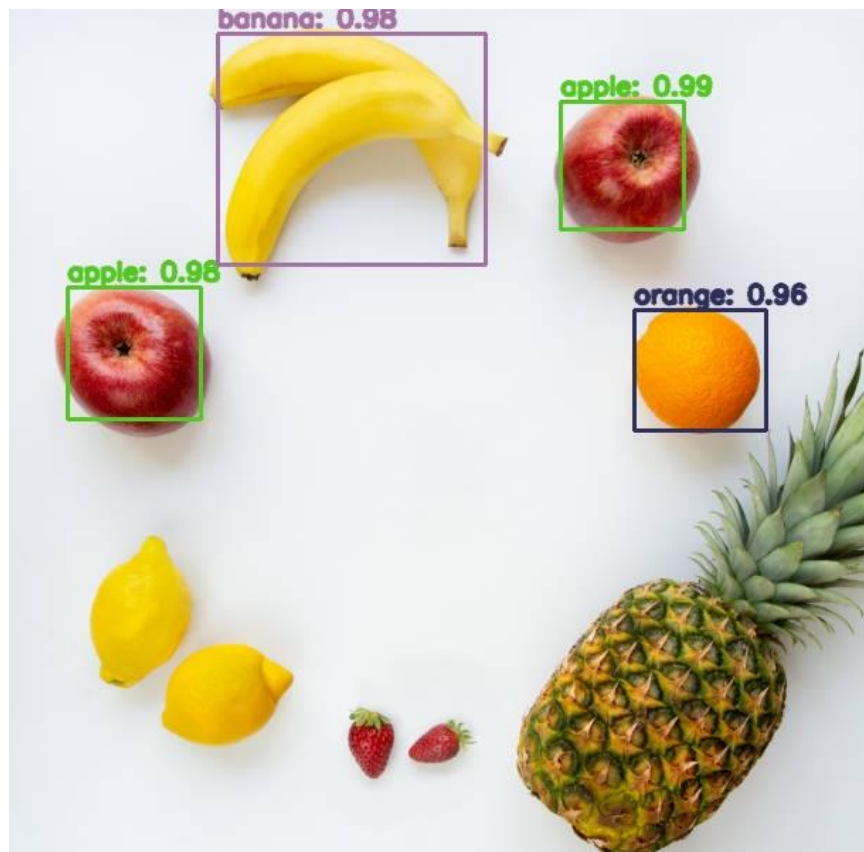


Figura 5.8. Detección en imagen. Varios objetos, distintas clases.

Dentro de las características de la aplicación destaca la versatilidad, ya que como se mostró en el presente trabajo, para reconocer otra categoría basta con recolectar y etiquetar nuevas imágenes (por ejemplo, de producto bueno y malo, acorde a los estándares de calidad que se requieran), entrenar el modelo de red neuronal y utilizar los pesos resultantes en la aplicación.

## VI. CONCLUSIONES

En los últimos años se han presentado avances importantes en el campo de la visión artificial, es el caso de modernos modelos de detección en tiempo real como YOLOv4, YOLOR, y YOLOX.

En el presente trabajo se realizaron pruebas para demostrar la utilidad que pueden tener en el área de la agricultura. Como ejemplo de su aplicación, se mostró el desarrollo de un programa minimalista y versátil para la detección de cuatro variedades de productos: manzana, naranja, plátano y jalapeño.

Los modelos lograron una media de precisión y velocidad de detección notables, por ello se podrían emplear en los procesos de control de calidad de la industria agrícola, por ejemplo, en la etapa de selección o separación del producto bueno con el malo. Resultando en una buena opción para contribuir al desarrollo sustentable del país, específicamente para ayudar a aumentar la productividad agrícola que se requerirá en las próximas décadas.

Para su aplicación en una empresa, se debe recolectar una gran cantidad de imágenes y/o extraer *frames* de videos en el entorno en el que se trabajara. Además, las imágenes deben ser de alta calidad para obtener mejores resultados en el entrenamiento del modelo.

**VII. BIBLIOGRAFÍA**

1. Aguilar-Alvarado, J., y Campoverde-Molina, M. (2020). Clasificación de frutas basadas en redes neuronales convolucionales. *Polo del Conocimiento*, 5(01), 3-22. <http://dx.doi.org/10.23857/pc.v5i01.1210>
2. Bochkovskiy, A., Wang, C.-Y., y Liao, H.-Y. M. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. <https://doi.org/10.48550/arxiv.2004.10934>
3. Chung, D. T. P., y Van Tai, D. (2019). A fruits recognition system based on a modern deep learning technique. *Journal of Physics: Conference Series*, 1327(1: 012050). <https://doi.org/10.1088/1742-6596/1327/1/012050>
4. FAO (2014). La alimentación en América Latina y el Caribe. Recuperado el 10 de diciembre de 2020, de <http://www.fao.org/3/a-i3592s.pdf>
5. FAO (2017). El futuro de la alimentación y la agricultura: Tendencias y desafíos. Recuperado el 15 de enero de 2021, de <http://www.fao.org/3/a-i6881s.pdf>
6. Figueroa, D. y Roa, E. (2016). Sistema de visión artificial para la identificación del estado de madurez de frutas (granadilla). *Revista Redes de Ingeniería*. 7(1), 84-92. <http://dx.doi.org/10.14483/udistrital.jour.redes.2016.1.a08>
7. Gongal, A., Amatya, S., Karkee, M., Zhang, Q., y Lewis, K. (2015). Sensors and systems for fruit detection and localization: A review. *Computers and Electronics in Agriculture*, 116, 8-19. <https://doi.org/10.1016/j.compag.2015.05.021>
8. Guillazca G., Carlos A., Hernández A., Valeria K. (2020). Clasificador de Productos Agrícolas para Control de Calidad basado en Machine Learning e Industria 4.0. *Revista Perspectivas*, 2 (2), 21-28. <https://doi.org/10.47187/perspectivas.vol2iss2.pp21-28.2020>



## BIBLIOGRAFÍA

9. Heras, D. (2017). Clasificador de imágenes de frutas basado en inteligencia artificial. *Revista Killkana Técnica*, 1(2), 21-30. [https://doi.org/10.26871/killkana\\_tecnica.v1i2.79](https://doi.org/10.26871/killkana_tecnica.v1i2.79)
10. Hongkun, T., Tianhai, W., Yadong, L., Xi, Q., y Yanzhou, L. (2020). Computer vision technology in agricultural automation —A review. *Information Processing in Agriculture*, 7(1), 1-19. ISSN 2214-3173, <https://doi.org/10.1016/j.inpa.2019.09.006>
11. Hossain, M. S., Al-Hammadi, M., y Muhammad, G. (2019). Automatic Fruit Classification Using Deep Learning for Industrial Applications. *IEEE Transactions on Industrial Informatics*, 15 (2), 1027-1034. <https://doi.org/10.1109/TII.2018.2875149>
12. IBM Cloud Education. (2020). Obtenido de IBM: <https://www.ibm.com/cloud/learn/deep-learning>
13. INEGI-SAGARPA (2015). Encuesta nacional agropecuaria 2014. *Conociendo el campo de México*. Recuperado el 15 de diciembre de 2020, de [https://www.inegi.org.mx/contenidos/productos/prod\\_serv/contenidos/espanol/bvinegi/productos/nueva\\_estruc/7028250\\_73923.pdf](https://www.inegi.org.mx/contenidos/productos/prod_serv/contenidos/espanol/bvinegi/productos/nueva_estruc/7028250_73923.pdf)
14. Jyoti Jhavar (2016). Orange Sorting by Applying Pattern Recognition on Colour Image. *Procedia Computer Science*, 78, 691-697. ISSN 1877-0509. <https://doi.org/10.1016/j.procs.2016.02.118>
15. kanbanize. (2022). Obtenido de kanbanize: <https://kanbanize.com/kanban-resources/getting-started/what-is-kanban>
16. Keras. (2022). Obtenido de Keras: <https://keras.io/about/>

## BIBLIOGRAFÍA

17. Megha, A., y Lakshmana (2016). Computer Vision Based Fruit Grading System for Quality Evaluation of Tomato in Agriculture industry. *Procedia Computer Science*, 79, 426-433. <https://doi.org/10.1016/j.procs.2016.03.055>
18. OpenCV team. (2022). Obtenido de OpenCV: <https://opencv.org/about/>
19. Parico, A. I. B., y Ahamed, T. (2021). Real Time Pear Fruit Detection and Counting Using YOLOv4 Models and Deep SORT. *Sensors*, 21(14): 4803). <https://dx.doi.org/10.3390/s21144803>
20. Payne, J. (2022). Obtenido de developer.com: <https://www.developer.com/languages/python-benefits>
21. Pérez Vázquez, A., Leyva Trinidad, D. A., y Gómez Merino, F. C. (2018). Desafíos y propuestas para lograr la seguridad alimentaria hacia el año 2050. *Revista Mexicana de Ciencias Agrícolas*, 9, 175-189. <https://doi.org/10.29312/remexca.v9i1.857>
22. Potdar, R. R. (2021). Obtenido de kaggle.com: <https://www.kaggle.com/raghavrpotdar/fresh-and-stale-images-of-fruits-and-vegetables>
23. Praharsha, V. (2021). Obtenido de OpenGenus IQ: <https://iq.opengenus.org/yolov4-model-architecture>
24. Raj, R., Nagaraj, S. S., Ritesh, S., Thushar, T. A., y Aparanji, V. M. (2021). Fruit Classification Comparison Based on CNN and YOLO. *IOP Conference Series: Materials Science and Engineering*, 1187(1: 012031). <https://doi.org/10.1088/1757-899X/1187/1/012031>

## BIBLIOGRAFÍA

25. Simplilearn. (2022). Obtenido de Simplilearn: <https://www.simplilearn.com/tutorials/deep-learning-tutorial/what-is-tensorflow>
26. Sucari León, R., Aroquipa Durán, Y..., Quina Quina, L. D..., Quispe Yapó, E..., Sucari León, A..., y Huanca Torres, F. A. s. (2020). Visión artificial en reconocimiento de patrones para clasificación de frutas en agronegocios. *PURIQ*, 2(2), 166-180. <https://doi.org/10.37073/puriq.2.2.76>
27. Madan, P., Madhavan, S. (2020). Obtenido de IBM Developer: <https://developer.ibm.com/learningpaths/get-started-with-deep-learning/an-introduction-to-deep-learning/>
28. Madhavan, S., Jones, T. (2021). Obtenido de IBM Developer: <https://developer.ibm.com/articles/cc-machine-learning-deep-learning-architectures/>