

# **Proyecto Final de Estudios:**

## **Optimización de Procesos de la Industria**

### **Maderera por medio de Visión Artificial**

**Autores:** Cárdenas, Facundo - Cereda, Mariano

**Tutora:** Dra. Ing. Selva Rivera

18 de Octubre de 2023

## **Abstract**

El presente informe describe el desarrollo de una aplicación para optimización de procesos en una empresa dedicada a la industria maderera. Se describe la problemática encontrada, los defectos del proceso productivo actual y, en base a ello, se desarrolló la solución mas óptima considerada. Específicamente, la aplicación permitió agilizar el proceso de conteo de palos almacenados como paquetes, el cálculo del diámetro promedio de cada paquete y la integración con el operario mediante el desarrollo de la interfaz adecuada. Para agilizar el proceso de conteo, se utilizó una red neuronal convolucional que responde a la arquitectura de YOLOv5, la cual cumplía con los requisitos necesarios para el caso de uso. El desarrollo consideró la obtención de los datos, el etiquetado, la obtención del modelo y su posterior entrenamiento y evaluación. Además, se implementó una API para la comunicación de la información de cada paquete procesado, permitiendo el acceso remoto y mejorando el proceso de control de stock de la empresa. Finalmente, se desplegó la aplicación en una *Single Board Computer* y se generó un primer prototipo para ser utilizado en la empresa.

# Índice

<b>1. Introducción</b>	<b>5</b>
<b>2. Estados del Arte</b>	<b>6</b>
<b>3. Descripción del Proceso a Optimizar y Problemática</b>	<b>7</b>
<b>4. Objetivos</b>	<b>10</b>
<b>5. Descripción del Sistema</b>	<b>11</b>
<b>6. Parte 1: Algoritmo de detección</b>	<b>12</b>
6.1. Selección del modelo: YOLO . . . . .	13
6.2. Principio de funcionamiento de YOLO . . . . .	14
6.2.1. Descripción de la Arquitectura utilizada: YOLOv5n . . . . .	18
6.2.2. Función de pérdidas . . . . .	21
6.3. Obtención de los datos y etiquetado. . . . .	22
6.4. Entrenamiento del modelo . . . . .	25
6.4.1. Obtención de cajas de anclaje. . . . .	25
6.4.2. Archivos de configuración del modelo. . . . .	26
6.4.3. Ejecución del entrenamiento . . . . .	29
<b>7. Parte 2: API y Base de datos</b>	<b>34</b>
7.1. Tecnologías y herramientas utilizadas. . . . .	35
7.2. Desarrollo de la API . . . . .	36
7.3. Definición de rutas . . . . .	37
7.4. Configuración de Máquina virtual en Azure . . . . .	37
7.5. Diagrama de secuencia. . . . .	37
7.6. Despliegue y ejecución de la API. . . . .	39
<b>8. Parte 3: Aplicación de Integración</b>	<b>40</b>
8.1. Interfaz . . . . .	40
8.2. Desarrollo de la App . . . . .	42
8.3. Cálculo del diámetro promedio . . . . .	45
8.4. Hardware utilizado . . . . .	45
8.4.1. Sensor de distancia HC-SR04 . . . . .	46
8.4.2. Cámara . . . . .	47
8.4.3. Pantalla Táctil LCD de 7 pulgadas . . . . .	47

8.5. Procedimiento de uso de la aplicación . . . . .	47
<b>9. Resultados</b>	<b>50</b>
<b>10. Conclusiones</b>	<b>54</b>
<b>11. Trabajos Futuros</b>	<b>55</b>
<b>12. Referencias</b>	<b>56</b>
<b>13. Anexos</b>	<b>57</b>

## **1. Introducción**

La industria maderera desempeña un papel fundamental en la economía de la provincia de Mendoza, particularmente en su relación con la vitivinicultura, una de las principales actividades económicas regionales. A pesar de su importancia, esta industria aún se apoya en métodos tradicionales que han demostrado ser laboriosos, intensivos en tiempo y propensos a errores humanos.

Uno de los procesos particularmente laboriosos que hemos observado en una PyME local es el conteo manual de palos o troncos de madera. Dado el volumen significativo de material que se maneja regularmente, esta actividad consume una cantidad considerable de tiempo y su precisión tiende a deteriorarse a medida que avanza la jornada del operario.

Por otro lado, en los últimos cinco años han surgido diversos enfoques que aprovechan la visión artificial para el reconocimiento de objetos en imágenes. Este avance tecnológico ofrece una oportunidad valiosa para la optimización de procesos dentro de la industria maderera.

Frente a esta problemática identificada, este proyecto presenta la implementación de una aplicación de visión artificial diseñada específicamente para automatizar y mejorar el proceso de conteo de palos en los fardos de troncos. Además, explora cómo esta herramienta puede utilizarse para optimizar el seguimiento del inventario de troncos y calcular una estimación precisa del diámetro promedio en grupos de paquetes contabilizados.

## 2. Estados del Arte

La detección y el conteo de objetos en entornos industriales y logísticos son tareas cruciales que han experimentado avances significativos gracias al desarrollo de algoritmos de visión artificial (VA). Esta tecnología se ha convertido en una herramienta valiosa en diversas aplicaciones, citando como ejemplos el reconocimiento de cabeza de trigo (Thuan, D et al.) o el conteo de barras de hierro (Li W. et al.).

En la actualidad, existen aplicaciones de conteo de objetos disponibles en el mercado. Sin embargo, es esencial destacar que, a pesar de su popularidad, estas aplicaciones presentan limitaciones significativas en entornos específicos. Una de las limitaciones clave de las mismas es que, al ser genéricas, no se logran adaptar totalmente a ambientes de detección específicos. Por otra parte, estas aplicaciones solo se encargan del conteo, pero no del reconocimiento del tamaño de los objetos ni de su implementación en campos diversos, ya que su uso es mediante un teléfono celular.

En otro orden, en la última década, se han notado grandes avances en la tecnología de sistemas embebidos. Estos avances han llevado al desarrollo de sistemas potentes, a pesar de su tamaño reducido. Esto ha permitido implementar sistemas con altos niveles de procesamiento en dimensiones limitadas, lo que ha redefinido la forma en que interactuamos con la tecnología en diferentes ámbitos. Específicamente, esos sistemas se han vuelto capaces de ejecutar algoritmos de Visión Artificial, que requieren altos niveles de procesamiento, en dimensiones sorprendentemente pequeñas.

Por todo lo previamente mencionado, la solución desarrollada busca resolver todas estas falencias en una sola aplicación, ejecutada dentro de una *Single Board Computer* (SBC): la detección de objetos precisa y adaptada para el caso de estudio, la determinación de su tamaño y la facilidad de uso por parte de los operarios.

### **3. Descripción del Proceso a Optimizar y Problemática**

El proceso productivo actual se realiza, casi en su totalidad, de forma manual y cuenta con los siguientes pasos:

1. Los palos provienen en camiones desde el norte de la República Argentina. Los mismos son descargados por medio de una cargadora o tractor acondicionado para la tarea.
2. Luego, se los clasifica manualmente según su tipo y diámetro. A la hora de clasificar según el diámetro, se considera un rango de valores indicado en centímetros. Por ejemplo, 8 cm a 10 cm, 10 cm a 12 cm, etc.
3. Posteriormente, se arman los paquetes de palos (también denominados *fardos*), donde los mismos se agrupan según su tipo y rango de diámetros. El proceso de armado de los paquetes es manual. En la figura 1 se puede observar un paquete terminado.
4. A continuación, se realiza un conteo (manual) de la cantidad de palos que contiene el paquete ya armado. La geometría del paquete se mantiene aproximadamente constante, por lo que el parámetro que cambia es la cantidad de palos por fardo, dependiendo del diámetro de los mismos.
5. Luego, se asocia un número de identificación a cada fardo. De esta manera, el número de palos de cada paquete es anotado en una libreta, la cual es trasladada a una oficina para la carga de los datos en la base de datos de la empresa.
6. Se apilan todos los paquetes de un mismo tipo. Para los paquetes que no deben ser impregnados químicamente, el proceso termina en esta etapa.
7. Para los fardos que deben ser impregnados el proceso continúa con su traslado, por medio de una cargadora, hacia la zona de impregnado químico. Durante este proceso, un químico a base de arsénico es inyectado en los palos dentro de una cámara a alta presión. Dicha presión depende del diámetro promedio de los palos de cada fardo. Por lo cual, previo a la impregnación, el técnico encargado de este proceso debe conocer este valor para la regulación de la presión durante el impregnado. Actualmente estos valores son estimados, ya que medir el diámetro promedio de los palos por paquete es una tarea que demanda mucho tiempo.



Figura 1: Ejemplo de paquete terminado.

Durante el análisis del proceso previamente explicado se notaron varios puntos a optimizar y mejorar, principalmente durante las etapas 4, 5 y 7. A continuación se listan los mismos.

- El conteo de palos es extremadamente lento, ya que la cantidad de palos que puede contener un paquete va desde 50 (para diámetros grandes) hasta valores mayores a 200 (para diámetros pequeños). Además, si la persona encargada del conteo presenta algún problema durante la realización de dicha tarea, se debe comenzar nuevamente.
- Al anotarse los valores en una libreta, los mismos pueden extraviarse fácilmente. De suceder esto, todo el registro de fardos se pierde.
- La libreta con la información detallada previamente debe ser trasladada por una persona hasta las oficinas (ubicadas a 150 metros de la zona de trabajo) para ser cargadas en la base de datos. En caso de existir algún valor escrito que no puede comprenderse,

la persona encargada de la carga de datos debe comunicarse nuevamente con la persona que contabilizó los palos para resolver el inconveniente. Este proceso esta sujeto a muchas fallas y, además, consume demasiado tiempo que puede ser optimizado.

- El parámetro del diámetro promedio de los palos por paquete utilizado durante la impregnación es completamente aproximado, ya que no se puede calcular dicho dato a costa de una gran pérdida de tiempo (se debería medir el diámetro de cada palo).

## **4. Objetivos**

A continuación se listan los objetivos para el proyecto:

1. Se espera obtener un prototipo funcional el cual pueda dar el diámetro y la cantidad de palos del fardo, buscando optimizar el tiempo de la etapa 4 del proceso mencionado anteriormente.
2. Con los datos previamente obtenidos, se busca mejorar al detalle el proceso de impregnación. Es decir, evitar el uso excesivo o falta del preservante químico.
3. Se espera lograr una comunicación con el servidor de la empresa para cargar la información y también crear copias locales de la información recolectada. De esta manera se busca evitar la pérdida de datos y agilizar los tiempos de la etapa 5 del proceso.
4. Obtener un prototipo rentable para una PyME local, mediante la búsqueda de los insumos más adecuados que maximicen el rendimiento y minimicen los costos.

## 5. Descripción del Sistema

En base la problemática planteada, se propuso un sistema que, durante su desarrollo, se subdividió en 3 partes:

- Selección del modelo de visión artificial mas adecuado, entrenamiento, puesta a punto y despliegue sobre una *Single Board Computer (SBC)* (Raspberry pi 4, modelo B).
- Desarrollo de interfaz gráfica para manejo de la aplicación por parte del operario.
- Desarrollo y despliegue de una API en un servidor remoto para guardado de la información recolectada por la aplicación y simplificación del acceso a los datos.

La interacción entre las partes mencionadas se puede observar de forma gráfica en la figura 2.

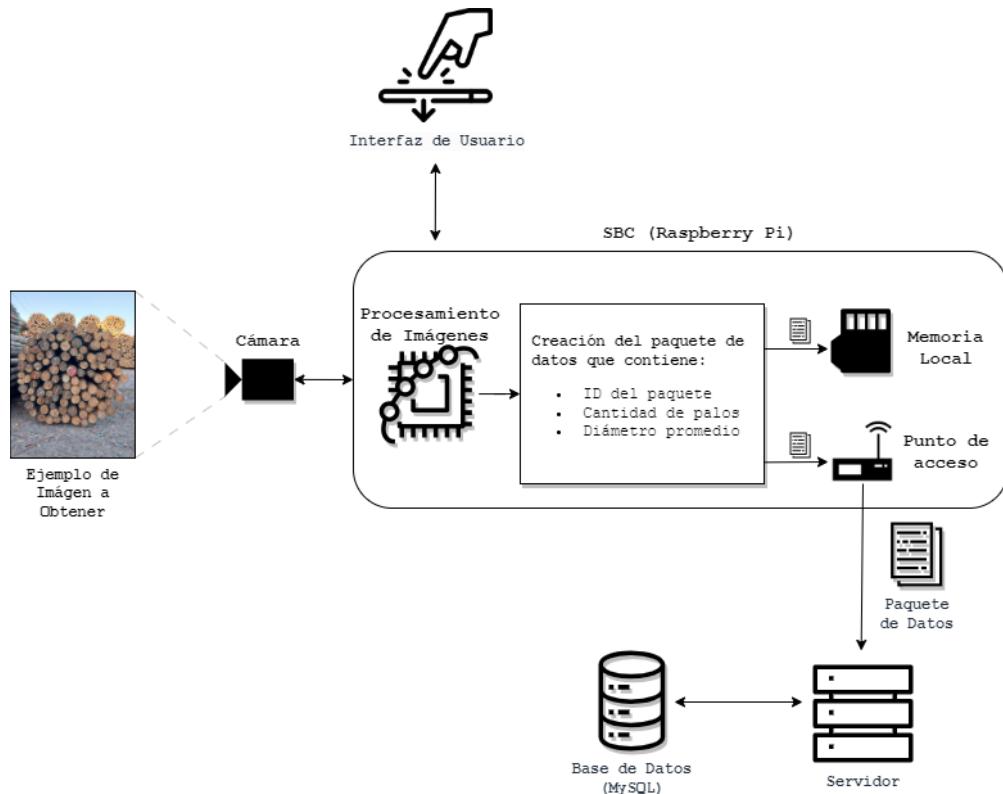


Figura 2: Ejemplo de paquete terminado.

Una vez visualizada la figura 2, se puede comprender mejor el flujo de trabajo que se explica a continuación:

El operario interactúa con la interfaz gráfica, la cual se muestra sobre una pantalla LCD *touch* de 7 pulgadas. En todo momento se puede observar la visual de la cámara y, cuando

el operario considere que la imagen es correcta y el paquete se visualiza claramente, puede proceder a tomar la foto mediante el botón correspondiente. Una vez obtenida, la aplicación procederá a detectar la cantidad de troncos visualizados haciendo uso del algoritmo YoloV5 (el cual se explicará mas adelante). Posteriormente, se le mostrará al operario, mediante la interfaz, la siguiente información:

- Cantidad de palos detectados
- Diámetro promedio
- Imagen donde se resalta cada uno de los palos detectados en el paquete. Al visualizar esta imagen, se puede apreciar si hubo un error en la detección o no.

Finalmente, el operario puede proceder a enviar la información del paquete detectado al servidor de la empresa. Para ello, debe ingresar el número de paquete (o ID) en la aplicación y luego presionar el botón correspondiente para enviar los datos en el servidor. Paralelamente, se realiza un copia local de la información obtenida.

## 6. Parte 1: Algoritmo de detección

El algoritmo de detección es la parte principal de la aplicación, utilizado para reconocer los palos, su tamaño y ubicarlos en la imagen.

Esta sección se divide en 4 partes:

1. Selección del algoritmo más óptimo para el problema a resolver.
2. Obtención de los datos y etiquetado.
3. Entrenamiento del modelo.
4. Evaluación de resultados.

Por ello, para seleccionar el modelo más adecuado, primero se tuvo comprender la problemáticas mas en detalle.



Figura 3: Paquete de palos terminado para su posterior detección.

Considerando el ejemplo de paquete mostrado en la figura 3 y lo mencionado en la sección 4, se concluyó que el algoritmo de detección de palos debía cumplir con los siguientes requisitos.

- Ser capaz de detectar múltiples cantidades de un mismo objeto en una misma imagen.
- No solo clasificar o reconocer un objeto con alta precisión, sino que también conocer su tamaño.
- Realizar la detección rápidamente, para que la aplicación cumpla con uno de los objetivos principales, que es reducir el tiempo de conteo de palos.
- Ser capaz de ser ejecutado en una *Single Board Computer*.

### 6.1. Selección del modelo: YOLO

Al inicio de esta sección se mencionaron los requerimientos que debe cumplir la aplicación. Luego, se mostraron los datos que fueron proporcionados para alimentar el modelo. En base a esta información, se decidió optar por un algoritmo de detección de objetos que permita,

no solo clasificar, sino también, realizar múltiples detecciones de objetos en una sola imagen, obtener su posición y reconocer su tamaño.

El algoritmo seleccionado para esta aplicación es denominado **YOLOv5**. Es la quinta versión de **YOLO**(Joseph R. et al, 2016), que por sus siglas en inglés significa *you only look once* o bien ”solo mira una vez”, haciendo referencia a que se realizan detecciones de objetos en una sola pasada a través de una imagen por la red neuronal, en contraposición a enfoques anteriores que requerían múltiples pasadas o etapas para detectar. La versión 5n de YOLO presenta múltiples ventajas que llevaron a optar por este modelo como el indicado para esta aplicación:

- Rapidez de la detección.
- Presenta una arquitectura ”nano”, que brinda una alta precisión y una poca cantidad de parámetros (tamaño pequeño en MB), siendo esto fundamental para ser ejecutado en una SBC con recursos limitados.
- Múltiples capas de detección, permitiendo detectar de forma eficaz palos de diferentes tamaños.
- Utilizado previamente en casos similares de paquetes de objetos pequeños. Por ejemplo, fue utilizado para la detección de barras de hierro en paquetes (Li W. et al).
- Código abierto para su acceso y entrenamiento de modelos personalizados.
- Extensa documentación.

## 6.2. Principio de funcionamiento de YOLO

Antes de comenzar con el desarrollo del principio de funcionamiento del proceso de detección, se deben comprender las siguientes definiciones:

- **CNN**: Las redes neuronales convolucionales son un tipo de arquitectura de red neuronal profunda especialmente diseñada para procesar imágenes (entre otras). Para este caso, se utilizan para extraer características importantes de la imagen que se utilizarán para la detección de objetos.
- **Cajas delimitadoras (Bounding Boxes)**: Son rectángulos que rodean a los objetos detectados en una imagen. Estas cajas se utilizan para definir la ubicación y el tamaño de los objetos detectados.

- **Cajas de verdad (Ground Truth Boxes):** Son las cajas delimitadoras que se consideran como la verdad absoluta en un conjunto de datos etiquetado. Estas cajas son obtenidas durante el etiquetado de objetos y se utilizan durante el entrenamiento.
- **IoU (Intersection Over Union):** Es una métrica que se utiliza para evaluar cuán precisas son las predicciones de las *bounding boxes* en comparación con las *ground truth boxes*. Se calcula como el área entre las cajas delimitadora predichas y las cajas de verdad dividida por la unión de ambas. En la figura 6 se puede observar la ecuación a aplicar.
- **Supresión de no máximos (Non-Maximum Suppression):** Es un proceso de filtrado utilizado para eliminar las detecciones duplicadas o superpuestas y mantener solo las detecciones más confiables en una imagen.
- **Cajas de anclaje (anchor boxes):** Son cajas con tamaños y relaciones de aspecto específicas que son utilizadas por la red para incorporar el conocimiento de los datos de entrenamiento en el modelo para ayudar a realizar mejores predicciones.

Una vez comprendidos estos conceptos, se puede explicar el proceso de detección utilizado por YOLO.

El proceso comienza con una imagen tomada como entrada y procesada por una red neuronal convolucional (CNN). La arquitectura de esta CNN se explicará en la siguiente sección, y es lo que va cambiando mayormente a lo largo de las diferentes versiones de YOLO. La red elegida cuenta con 3 etapas de detección, donde las 3 se comportan de igual manera, pero cada una de ellas reduce el tamaño de la imagen de entrada a diferentes escalas.

Cada etapa de detección de la red comienza dividiendo la imagen en una cuadrícula con un tamaño  $S \times S$  (figura 5 (a)) y realiza predicciones de cajas delimitadoras en cada celda de la cuadrícula (figura 5 (b)). Las cajas de anclaje ayudan a refinar estas predicciones para diferentes tamaños de objetos. Luego, cada celda de la cuadrícula realiza múltiples predicciones de cajas, junto con puntuaciones de objeto que indican la probabilidad de que haya un objeto en la caja, y predicciones de clase que identifican la categoría del objeto (figura 5 (c)).

Finalmente, se aplica un umbral a las puntuaciones de objeto para determinar cuáles cajas son válidas, y luego se realiza una supresión de no máximos para el filtrado de detecciones. Finalmente, el resultado es una lista de objetos detectados con sus ubicaciones y etiquetas de clase.

Por otro lado, las predicciones del modelo del tamaño y localización de las cajas delimitadoras,  $tx$ ,  $ty$ ,  $tw$  y  $th$ , (salidas de la CNN) son adaptadas considerando los valores de las cajas de anclaje. Así, las *bounding boxes* finales se obtienen según las siguientes ecuaciones:

$$box_x(bx) = (2 * \sigma(tx) - 0,5) + c_x$$

$$box_y(by) = (2 * \sigma(ty) - 0,5) + c_y$$

$$box_{width}(bw) = p_w * (2 * e^{t_w})^2$$

$$box_{height}(bh) = p_h * (2 * e^{t_h})^2$$

Donde  $p_w$  y  $p_h$  son el ancho y el alto de las *anchor boxes* correspondientes y  $c_x$ ,  $c_y$  son las coordenadas del origen de la célula de cuadrícula encargada de la detección (ver figura 4). Así, se obtiene el vector  $\mathbf{y}$  , que es la representación final de la caja delimitadora.

$$\mathbf{y} = [po, box_{x-coordinate}, box_{y-coordinate}, box_{height}, box_{width}, p_{class}] \quad (1)$$

Donde:

- **po**: corresponde a la probabilidad de que la cuadrícula contenga un objeto. En la figura 5 (c), se pueden observar en color amarillo aquellas cuadriculas donde se tiene un valor de pc elevado.
- **box-x-coordinate**: Coordenada x del centro de la caja delimitadora.
- **box-y-coordinate**: Coordenada y del centro de la caja delimitadora.
- **box-height**: Alto de la caja delimitadora.
- **box-width**: Ancho de la caja delimitadora.
- **p-class**: Indica la clase a la que corresponde el objeto.

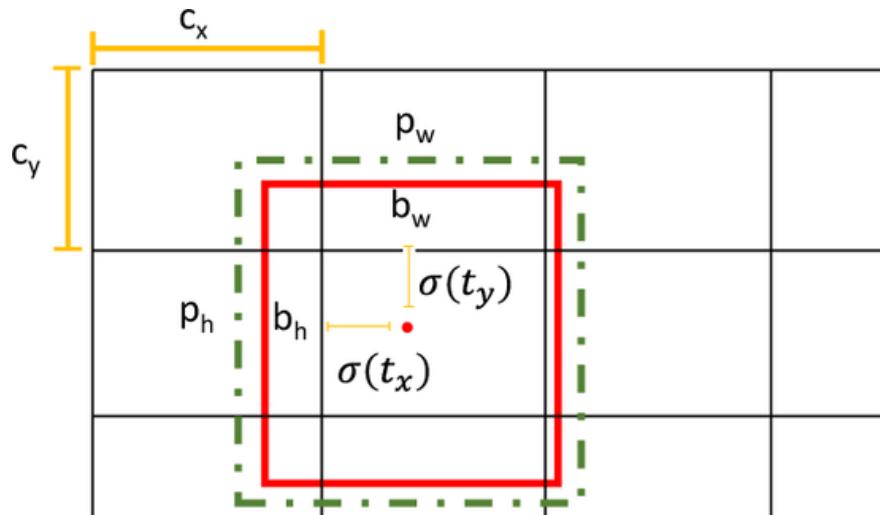


Figura 4: Puntos de predicción de cajas delimitadoras (recuadro de línea rellena) en base a cajas de anclaje (recuadro de línea punteada).

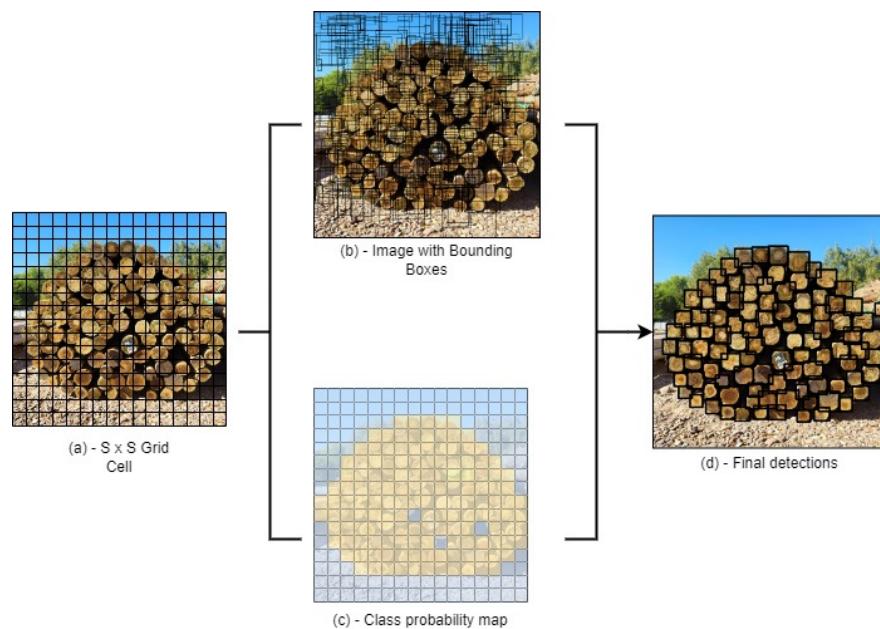


Figura 5: Esquema de funcionamiento de YOLO.

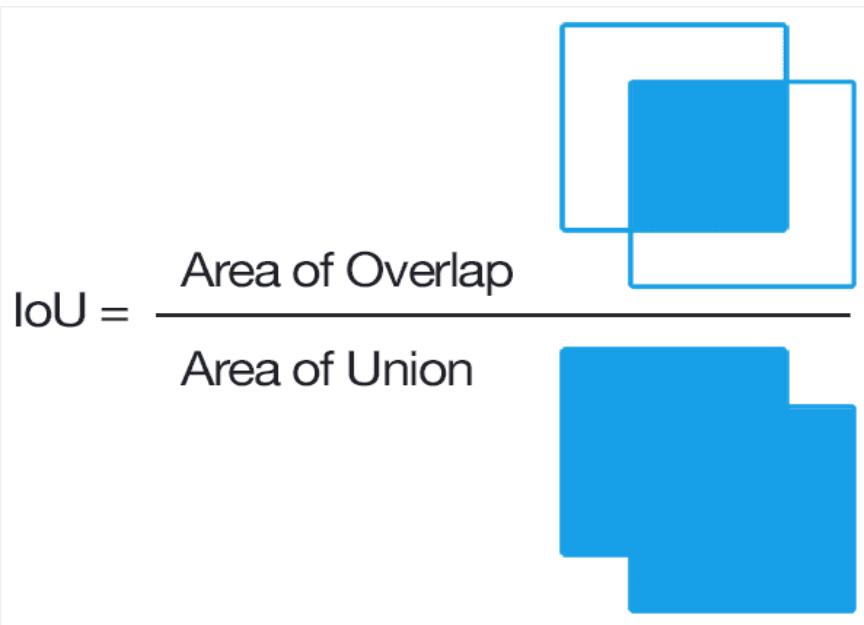


Figura 6: Ecuación e ilustración de IoU.

Finalmente, se eliminan aquellas detecciones con un valor de confianza (po) debajo de un umbral y se aplica la supresión de no máximos para eliminar aquellas predicciones duplicadas o superpuestas. Este proceso se realiza de la siguiente manera.

- Se obtienen todas las detecciones de objetos y se ordenan por su confianza (po) en orden descendente.
- Se selecciona la detección más confiable y se agrega a la lista de detecciones finales.
- Se calcula la superposición entre la caja delimitadora de la detección actual y las cajas delimitadoras de las detecciones restantes.
- Si el solapamiento entre la detección actual y otra detección es mayor que un umbral predefinido, se elimina la detección con la confianza más baja, ya que se considera un duplicado.
- Este proceso se repite para todas las detecciones en orden de confianza, lo que resulta en una lista final de detecciones no duplicadas y confiables.

#### 6.2.1. Descripción de la Arquitectura utilizada: YOLOv5n

La arquitectura de red neuronal utilizada es la de YOLOv5n (o bien, YOLOv5 nano). La misma es la versión de menor tamaño de YOLOv5 y se seleccionó la misma ya que combina la precisión de YOLOv5 con un tamaño de red pequeño, permitiendo que el proceso de

detección sea rápido y que pueda ser aplicado en una SBC, que cuenta con recursos computacionales limitados.

La red específicamente se divide en 3 secciones y 25 capas (figura 7).

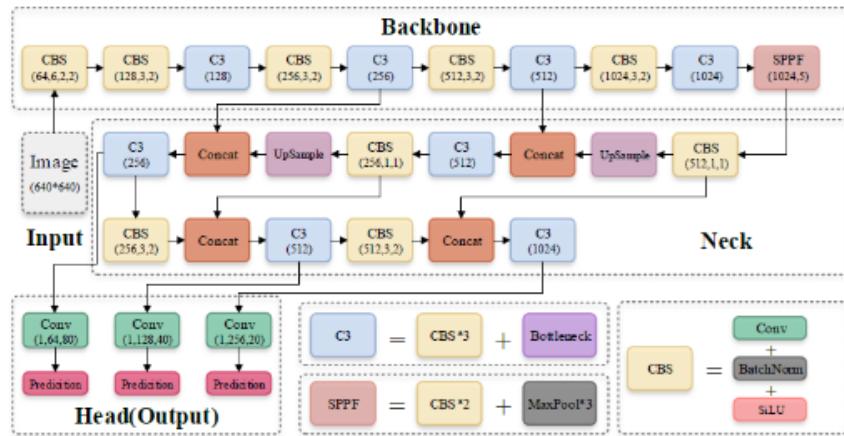


Figura 7: Arquitectura de red de YOLOv5n.

Las 3 secciones se listan y explican a continuación:

- **Espina Dorsal (Backbone):** Es la parte inicial de la CNN que se encarga de extraer características de la imagen de entrada. En YOLOv5, se utiliza una arquitectura de red llamada **CSPDarknet53** como espina dorsal. El propósito del es reducir gradualmente la resolución espacial de la imagen de entrada mientras se aumenta la profundidad de las características extraídas.
- **Cuello (Neck):** Sigue a la espina dorsal y tiene como objetivo fusionar y combinar las características de diferentes resoluciones espaciales para mejorar la detección de objetos a múltiples escalas. Se utiliza un cuello llamado **PANet (Path Aggregation Network)**. Toma las características de diferentes niveles de resolución generadas por el *backbone* y las fusiona para ser utilizadas en la etapa de detección final.
- **Head (cabeza):** Se encarga de realizar las predicciones finales de detección de objetos. En este caso, se compone de tres capas convolucionales que predicen el vector de la caja delimitadora (ecuación 1). Aquí se utiliza la supresión de no máximos (previamente mencionada). Al existir 3 capas de detección, permite detectar palos (o el objeto determinado) de diferentes tamaños y escalas.

A su vez, se puede observar que en la figura 7, diferentes bloques individuales conforman la arquitectura de la red. Cada uno de ellos (CBS, C3, Bottleneck y SPPF) se pueden apreciar detalladamente en las figuras 8 y 9. Es importante aclarar que el Bloque **CBS** esta

presente en toda la red y no solamente cuando se encuentra dentro de C3. Este bloque es fundamental, ya que las diferentes versiones de YOLOv5 lo que hacen es repetir una mayor o menor cantidad de veces los bloques **Bottleneck** dentro de **C3** según dos parámetros de entrada. Esos parámetros son **width multiple** y **depth multiple** y para este caso tienen los valores de **0.33** y **0.25** respectivamente. Teniendo en cuenta que esta es la versión con menor cantidad de parámetros, la mayor cuenta con valores de **1.33 (depth multiple)** y **1.25 (width multiple)**.

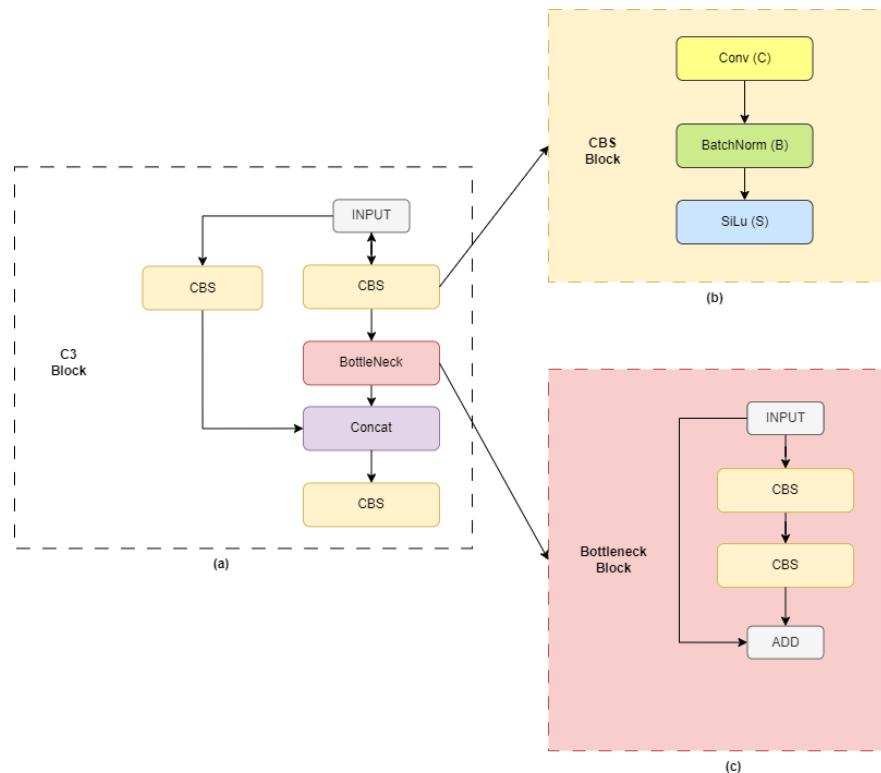


Figura 8: Bloque C3 (a), CBS (b) y Bottleneck (c).

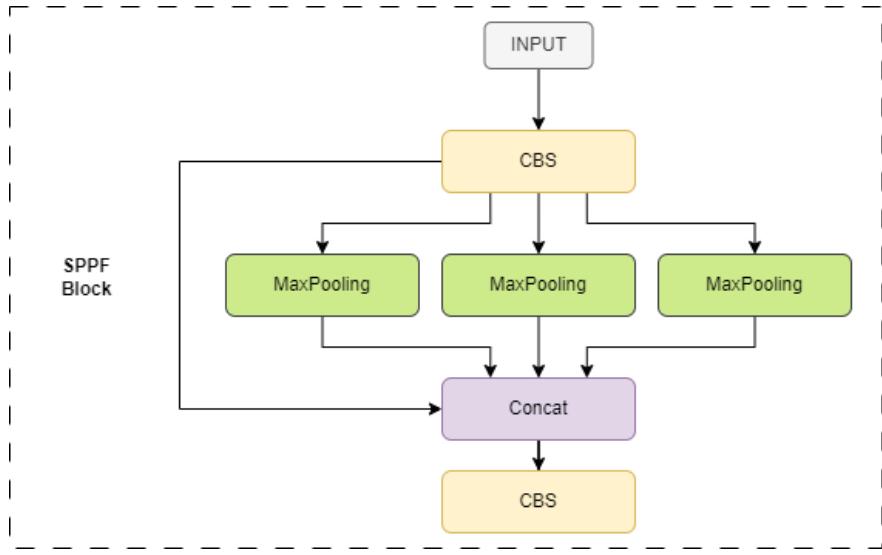


Figura 9: Bloque SPPF.

### 6.2.2. Función de pérdidas

La función de pérdida en *deep learning* es una métrica matemática que cuantifica la diferencia entre las predicciones de un modelo y los valores reales del conjunto de datos. Se utiliza para guiar el proceso de optimizar el modelo durante el entrenamiento para minimizar los errores.

A diferencia de otros algoritmos de detección o clasificación, en este caso se calcula una pérdida compuesta. Esta composición está formada por 3 tipos de pérdidas individuales:

- **Pérdidas de caja (*box loss*)**: Cuantifica la diferencia entre las coordenadas de las cajas delimitadoras predichas y las coordenadas de la caja delimitadora real.
- **Pérdidas de objeto (*object loss*)**: Mide la diferencia entre la confianza (probabilidad de que un objeto esté presente en la caja delimitadora) predicha por el modelo y la confianza real del objeto en el conjunto de datos.
- **Pérdidas de clase (*class loss*)**: Esta pérdida mide la discrepancia entre las predicciones de clase y las etiquetas reales. En nuestro caso no es tan relevante, ya que solo necesitamos detectar una sola clase.

En base a estas 3 variables se obtiene una fórmula que pondera cada una de estas pérdidas mediante 3 parámetros *lambda*. Cada uno de los cuales permite asignar diferentes pesos o relevancias a las funciones de pérdidas individuales. En la ecuación 2 se puede apreciar la expresión final.

$$Loss = \lambda_{box-loss} * L_{box} + \lambda_{obj-loss} * L_{obj} + \lambda_{cls-loss} * L_{cls} \quad (2)$$

### 6.3. Obtención de los datos y etiquetado.

Estos dos primeros pasos se centraron en identificar, recopilar y analizar los conjuntos de datos que ayudaron a alcanzar los objetivos del proyecto.

Para este caso, los datos fueron imágenes de palos individuales y paquetes que luego alimentaron el entrenamiento del modelo. En total, se contó con 258 imágenes de palos para el entrenamiento (considerando con y sin aumentado de datos).

Tipo	Nº imágenes (+ Aumentado)	Cantidad de palos	Tamaño (píxeles)
Entrenamiento	258	15598	640 x 640

Cuadro 1: Cantidad de imágenes de entrenamiento

Se consideró ese numero total de imágenes para luego obtener un 15 % para validación y 10 % para test. A continuación (figura 10) se pueden observar algunos ejemplos de las imágenes utilizadas durante el entrenamiento del modelo.



Figura 10: Ejemplos de imágenes utilizadas para el entrenamiento del modelo.

La técnica de aumentado de datos permite generar nuevas instancias de datos a partir de los originales mediante diversas transformaciones o variaciones. El objetivo principal (en este caso) es aumentar la cantidad de imágenes de entrenamiento disponibles. En la figura 11 se puede apreciar un ejemplo de 2 operaciones de aumentado: rotación y zoom. De esta forma, se pasa de tener solo una imagen (figura 11 (a)) a contar con 4.

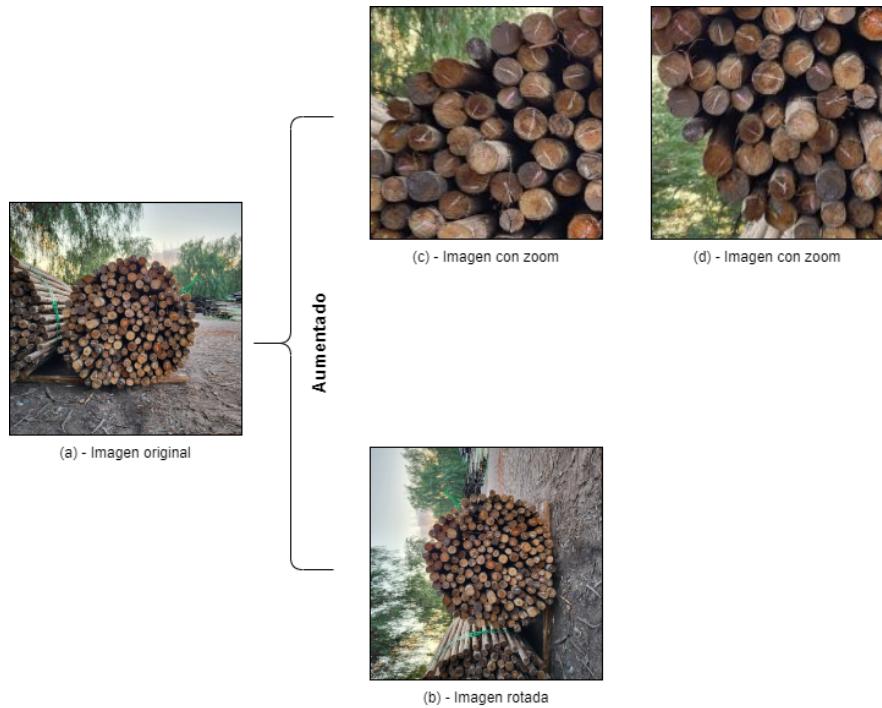


Figura 11: Foto antes (a) y despues (b),(c),(d) de las operaciones de aumentado.

Además, se agregaron imágenes que no cuentan con ningún palo. De esta manera, el algoritmo puede lograr descartar patrones que pueden ser confusos.



Figura 12: Imágenes sin ningún palo a detectar utilizadas durante el entrenamiento.

Para finalizar la etapa de preparación de los datos se debe realizó el etiquetado. El cual se explica a continuación.

El algoritmo de YOLOv5 necesita 2 archivos para el entrenamiento por cada imagen:

- La propia imagen, en un formato ".jpg", ".png", ".jpeg", etc.

- Un archivo en formato ".txt", que contiene la información de cada objeto etiquetado en la foto.

El archivo ".txt" del etiquetado debe tener el siguiente formato, denominado YOLO:

```
<clase-objeto> <x-centro> <y-centro> <ancho> <alto>
```

Listing 1: Formato de etiquetado YOLO

Donde:

- **clase-objeto**: es el índice de la clase del objeto. Para este caso solo se tiene el índice "0", que se corresponde con la clase de *palos*.
- **x-centro**: es la coordenada x del centro de la caja delimitadora del objeto.
- **y-centro**: es la coordenada y del centro de la caja delimitadora del objeto.
- **ancho**: es el ancho de la caja delimitadora del objeto.
- **alto**: es la altura de la caja delimitadora del objeto.

En la figura 13 se puede apreciar un ejemplo de archivo de etiquetado, que se corresponde con una imagen que contiene 10 palos. También se puede observar que todas las coordenadas y dimensiones de los objetos se normalizaron al tamaño de la imagen, por lo cual los valores se encuentran entre 0 y 1. Por otro lado, es importante aclarar que la imagen y el archivo de etiquetas deben tener el mismo nombre para que el algoritmo reconozca la correspondencia entre ambos durante el entrenamiento.

0 0.350781 0.527344 0.042188 0.035937
0 0.375000 0.569531 0.040625 0.042188
0 0.344531 0.577344 0.029687 0.029687
0 0.321094 0.602344 0.042188 0.042188
0 0.366406 0.607812 0.035937 0.031250
0 0.337500 0.642969 0.037500 0.032813
0 0.371875 0.653125 0.034375 0.034375
0 0.389062 0.698438 0.037500 0.037500
0 0.450000 0.713281 0.034375 0.035937
0 0.420312 0.692969 0.034375 0.035937

Figura 13: Ejemplo de archivo de etiquetas, cuya imagen relacionada contiene 10 palos.

Para realizar este etiquetado se utilizó **labelImg**, que es una herramienta de código abierto utilizada para etiquetar objetos en imágenes. Cuenta con una interfaz gráfica sencilla

y permite generar los archivos de texto en formato YOLO. A continuación, se muestra un ejemplo de como luce la interfaz que presenta esta aplicación. Es importante aclarar que se debe ser cuidadoso al recuadrar los palos manualmente, ya que estos recuadros servirán para generar las cajas de verdad durante el entrenamiento y de ello dependerá la calidad del modelo.



Figura 14: Interfaz de la aplicación labelImg para etiquetado de palos.

## 6.4. Entrenamiento del modelo

Para el entrenamiento se utilizó uno de los modelos disponibles de YOLOv5, que responde a la arquitectura explicada en la sección 6.2.1. El modelo que se utilizó es de código abierto y fue desarrollado en **python** con la librería **pytorch**.

### 6.4.1. Obtención de cajas de anclaje.

Una vez seleccionado el modelo a utilizar, se comenzó con el entrenamiento. En la sección 6.3.2 se mencionaron los datos a ser utilizados para entrenar el modelo y su formato. El paso siguiente fue seleccionar las *anchor boxes* considerando la base de datos de imágenes etiquetadas para entrenar. Para ello, se implementó y ejecutó el algoritmo de *clustering K-means*, donde cada punto dato corresponde a una *ground truth box* del conjunto de imágenes de entrenamiento con ancho y alto definido. Además, a diferencia de un K-means tradicional, la métrica utilizada es el *IoU* entre dos cajas y no la distancia euclidiana. El resultado (figura 15) fueron 9 puntos, cuyas coordenadas x e y corresponden al alto y ancho (medido en píxeles) de las *anchor boxes* de entrada.

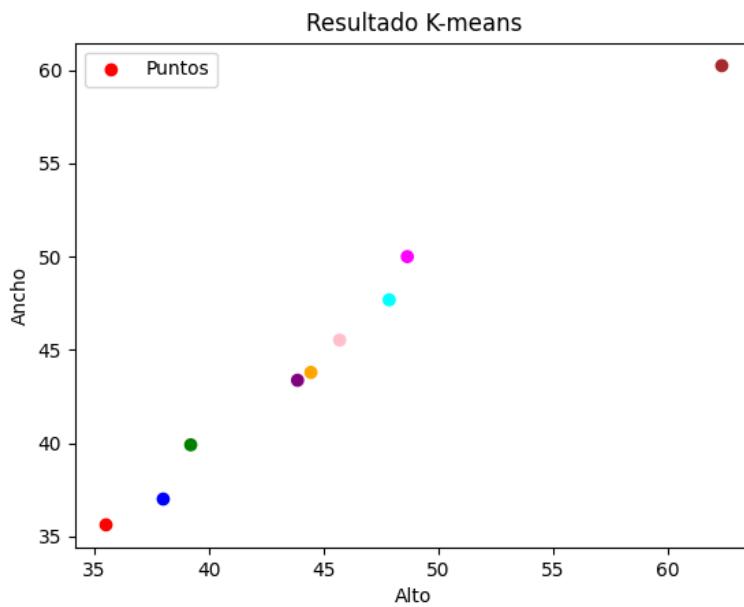


Figura 15: Centros encontrados de algoritmo K-means.

Teniendo en cuenta que los tamaños de las imágenes utilizadas son de 640 x 640 píxeles, los puntos mostrados en la figura 15 están escalados para ese tamaño.

#### 6.4.2. Archivos de configuración del modelo.

Posteriormente, se completaron 2 archivos de configuración. El primero contiene, principalmente, la configuración de la arquitectura de la red y las *anchor boxes* calculadas previamente. A continuación se muestra un ejemplo simplificado:

## Archivo de configuración de NN

```
nc: 1
depth_multiple: 0.33
width_multiple: 0.25
anchors:
  - [35.5044, 35.6166, 38.0, 37.0003, 39.2001, 39.9080]
  - [43.8585, 43.3709, 44.4421, 43.7912, 45.6977, 45.5243]
  - [47.8571, 47.6798, 48.6476, 50.0001, 62.3678, 60.2380]
backbone:
  [[-1, 1, Conv, [64, 6, 2, 2]], # 0-P1/2
   ...
  ]
head:
  [[-1, 1, Conv, [512, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 6], 1, Concat, [1]], # cat backbone P4
   [-1, 3, C3, [512, False]],
   ...
  ]
```

Donde:

- **nc** : Cantidad de clases de objetos, en este caso es 1 ya que solo se desea detectar palos.
- **depth multiple y width multiple**: Indica la profundidad de capas de la red neuronal. Esto difiere para las distintas versiones de YOLOv5 (m,s,l,n,x). En esta versión **nano**, se cuenta con valores de profundidad bajos y mencionados en la sección 6.2.1.
- **anchors**: Son las *anchor boxes* previamente obtenidas mediante K-means y aquí indicadas como listas. Cada lista corresponde a un grupo de 3 cajas de anclaje utilizadas para cada nivel de detección.
- **backbone**: Es una lista de listas utilizada para establecer la configuración de la primera sección de la CNN. Para fines prácticos solo se muestra una fila, pero esto se repite para cada capa de la espina dorsal de la red. Cada fila contiene la siguiente información:
  - -1: Indica que la entrada proviene de la capa anterior de la red.
  - 1: Indica que se realizará una sola operación de convolución:
  - Conv: Especifica que se utilizará una capa convolucional.

- (64, 6, 2, 2): Es una lista de argumentos para la capa convolucional que incluye el número de canales de salida (64), el tamaño del kernel (6x6), el paso (stride) en horizontal (2) y vertical (2).
- **head**: De forma similar al *backbone*, solo que en este caso se agregan nuevas operaciones como concatenaciones (operación que conecta zonas no contiguas de la red) y muestreos ascendentes (aumento de la resolución). Por ejemplo, la información contenida en la tercer fila es sobre una operación de concatenación:
  - (-1, 6): Indica una concatenación de la entrada actual con una capa anterior (en este caso, la capa 6 del "backbone" P4).
  - 1: Indica que se realizará una sola operación de convolución.
  - Concat: Especifica la operación de concatenación de capas no contiguas.
  - (1): Especifica el eje a lo largo del cual se realizará la concatenación. En este caso, [1] indica que la concatenación se realizará a lo largo del eje de canales (o "profundidad" del mapa de características).

El segundo archivo de configuración posee la información de los parámetros de entrenamiento. A lo largo de distintas pruebas se logró determinar que los parámetros para los que se obtuvieron mejores resultados son los mostrados en la tabla 2.

Parametro	Valor	Descripción
Learning Rate	0,01	Tasa de aprendizaje.
Momentum	0,937	Utilizado por el optimizador ( <i>optimizer</i> ).
box loss	0,01	Coeficientes de ponderación para las pérdidas de cajas. (Ver ecuación 2)
cls loss	0,05	Coeficientes de ponderación para las pérdidas de clase. (Ver ecuación 2)
obj loss	0,5	Coeficientes de ponderación para las pérdidas de regresión de objetos. (Ver ecuación 2)
iou threshold	0,5	Valor de umbral de IoU (Intersección sobre Unión) utilizado para determinar si una detección es verdadera positiva o falsa positiva durante el entrenamiento.
optimizer	Adam	Función que adapta los atributos de la red neuronal, como la velocidad de aprendizaje y los pesos.
epochs	150	Una sola pasada completa a través de todo el conjunto de datos de entrenamiento.
batch size	32	Es el número de muestras procesadas antes de actualizar el modelo.

Cuadro 2: Parámetros de entrenamiento

#### 6.4.3. Ejecución del entrenamiento

Para el entrenamiento de la red se utilizó una computadora con una **GPU NVIDIA GeForce RTX 3060 con 6GB de memoria RAM**. El proceso se ejecutó múltiples veces hasta obtener resultados satisfactorios para el caso de negocios tratado.

El resultado del entrenamiento fue guardado en un archivo denominado **weights.pt**, que contiene todos los pesos de la red obtenidos durante el entrenamiento y fue utilizado posteriormente para realizar las detecciones.

Para evaluar los resultados del modelo entrenado, se utilizaron diferentes métricas que se describen a continuación:

- **mAP0.5** (*mean average precision*): se calcula el mAP utilizando un umbral de IoU de 0.5 como criterio para determinar si una detección es considerada correcta o no. El valor de 0.5 significa que se requiere que al menos el 50 % de la región predicha por el modelo se superponga con la región real del objeto. En la figura 16 se puede observar

la evolución a lo largo del entrenamiento y la obtención de un valor final elevado de **97.4 %**.

- **mAP0.5-0.95** (*mean average precision*): se calcula el mAP promediando las puntuaciones de precisión y recuperación en un rango de umbrales de IoU que varían de 0.5 a 0.95 en incrementos pequeños. En la figura 17 se observa un valor final satisfactorio de **76.6 %**.
- **F1-Score:** Es útil para comprender cómo varía el rendimiento del modelo de detección a medida que se ajusta el umbral de confianza. En consecuencia, de esta curva (figura 18) se obtiene el valor de confianza utilizado para el filtrado de detecciones incorrectas o poco precisas. En este caso, el valor de confianza para el cual se obtiene un mejor resultado es **0.620**.
- **Precisión:** Es una métrica que mide la proporción de predicciones positivas (palos) que fueron correctas en comparación con todas las predicciones positivas realizadas por el modelo. Mide la capacidad del modelo para no etiquetar incorrectamente ejemplos negativos (fondo o *background*) como positivos. Responde a la ecuación:

$$P = \frac{Verdaderos_{positivos}(TP)}{Verdaderos_{positivos}(TP) + Falsos_{positivos}(FP)} \quad (3)$$

Se obtuvo un valor final elevado, de **98 %**.

- **Recuperación(*Recall*):** Mide la proporción de casos positivos reales que fueron correctamente detectados por el modelo en relación con el total de casos positivos reales. Es la capacidad del modelo para encontrar todos los casos positivos.

$$P = \frac{Verdaderos_{positivos}(TP)}{Verdaderos_{positivos}(TP) + Falsos_{negativos}(FN)} \quad (4)$$

Al igual que la precisión, el valor final es elevado y de **95 %**.

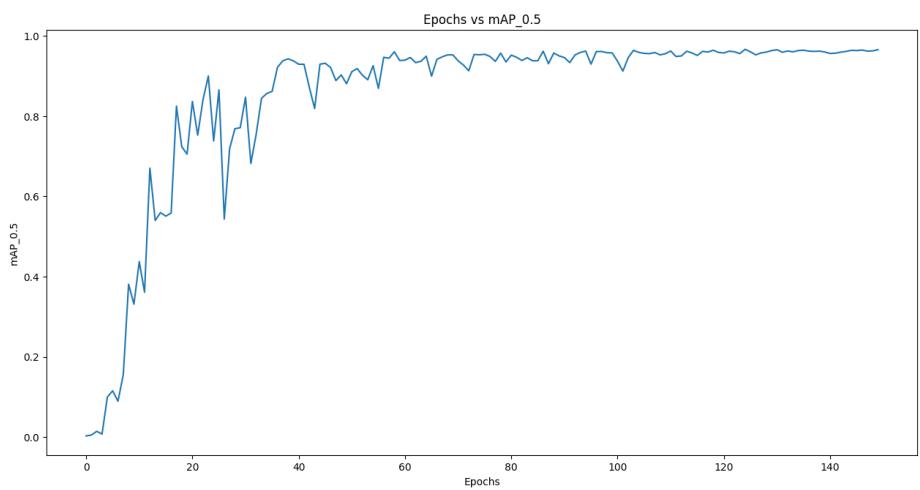


Figura 16: Resultados mAP 0.5.

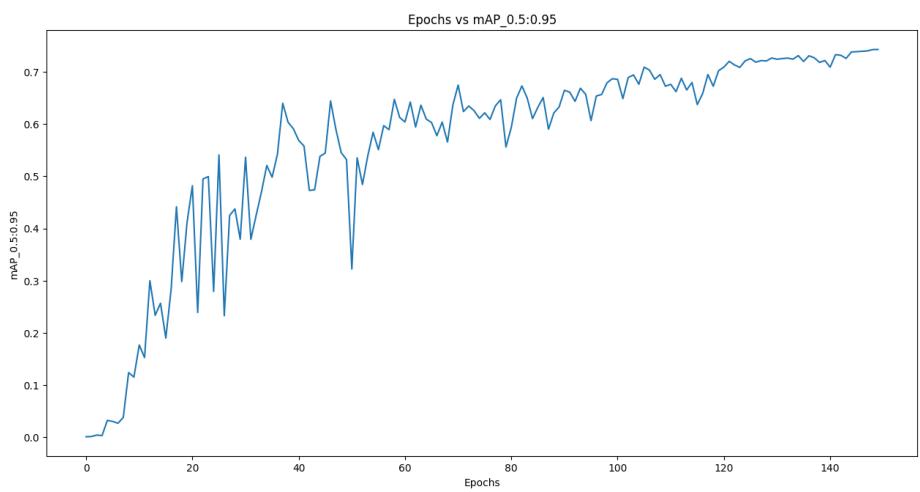


Figura 17: Resultados mAP 0.5:0.95

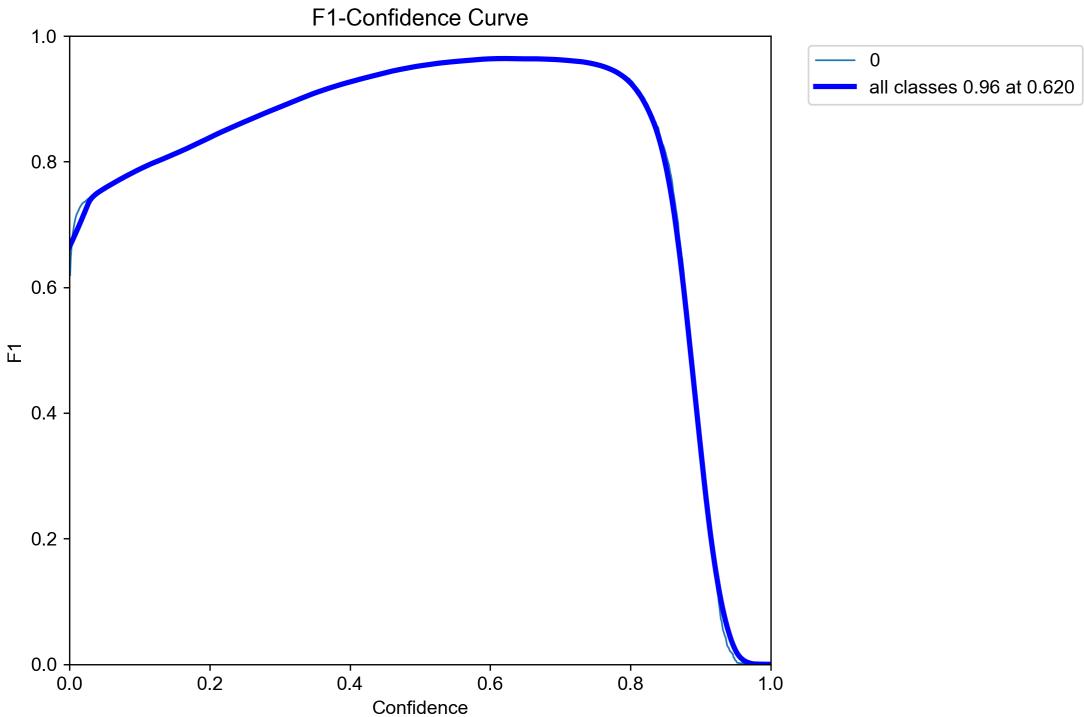


Figura 18: Curva F1 para resultados de entrenamiento.

Por último, una curva muy importante es la de **Precisión vs Recuperación (recall)**. Esta curva (mostrada en la figura 19) muestra el equilibrio entre precisión y recuperación para distintos valores de confianza. Un área elevada bajo la curva representa tanto un alto *recall* como una alta precisión. En este caso, se puede apreciar un área bajo la curva elevada, lo cual coincide con los altos resultados de precisión y recuperación obtenidos individualmente.

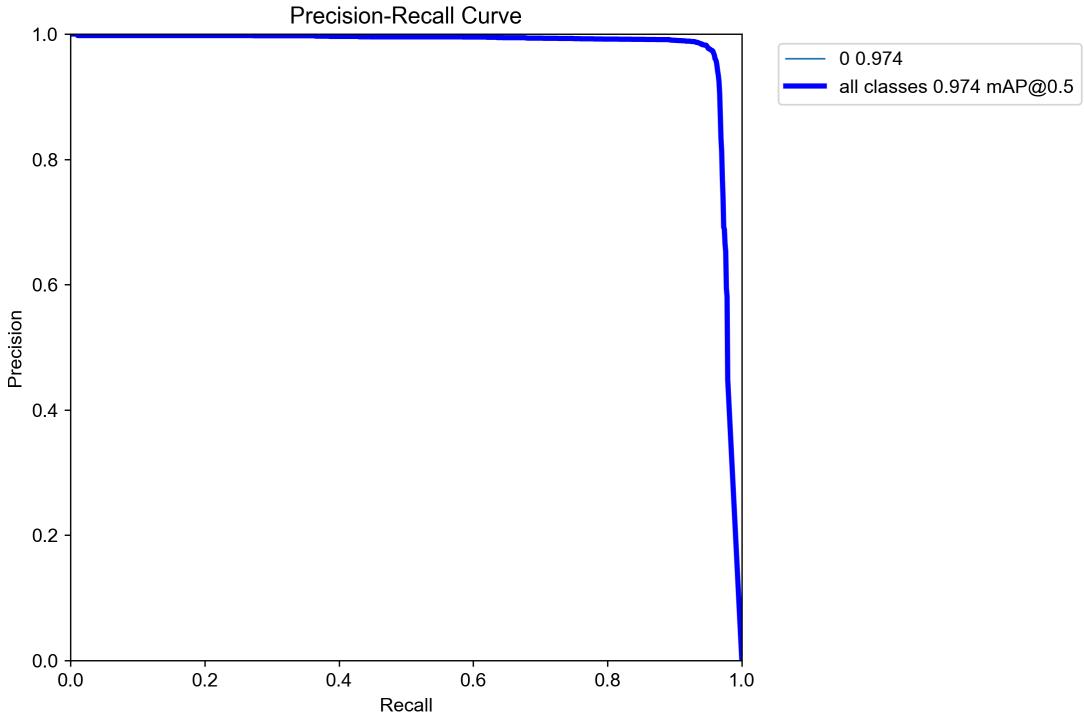


Figura 19: Curva Precision vs Recall.

Finalmente, se presentan imágenes que muestran los resultados obtenidos luego de ejecutar la detección en algunas de las imágenes del conjunto de pruebas o test.



Figura 20: Imagen resultante de la detección (derecha) y resultado de la detección con porcentaje de confianza (izquierda)(1).



Figura 21: Imagen resultante de la detección (derecha) y resultado de la detección con porcentaje de confianza (izquierda)(2).



Figura 22: Imagen resultante de la detección (3).

En la sección de anexos se podrán apreciar mayor cantidad de ejemplos de detecciones.

## 7. Parte 2: API y Base de datos

Toda la información obtenida durante el proceso de detección no es de utilidad si no es comunicada efectivamente al servidor de control de stock de la empresa. Para ello, se desarrolló una API que permite la comunicación entre la aplicación del sistema embebido,

encargada de la detección, y la base de datos que contiene la información de los paquetes.

Para el desarrollo y despliegue de la API se utilizaron las siguientes herramientas:

- Una máquina virtual de **Azure Cloud** para el despliegue en un ambiente productivo.
- **Nodejs** como motor de desarrollo y **javascript** como lenguaje de programación.
- **MySQL** como sistema de gestión de bases de datos para almacenar la información enviada a la API.

En la figura 23 se muestra un esquema de la interacción entre los diferentes módulos, con un mayor nivel de detalle.

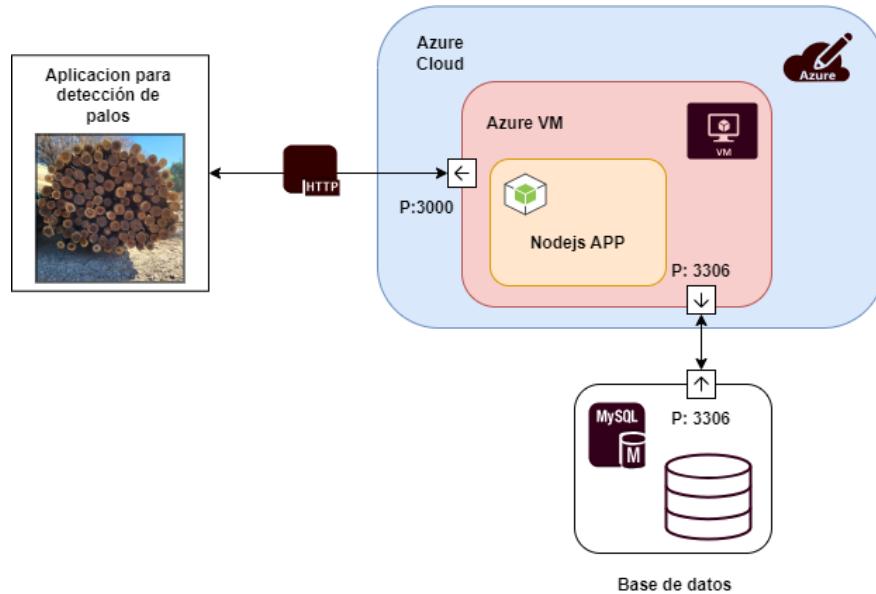


Figura 23: Detalle de comunicaciones con servidor.

### 7.1. Tecnologías y herramientas utilizadas.

Para el desarrollo de la API se utilizó **Nodejs**, que es un entorno de ejecución de código JavaScript del lado del servidor. Además, se utilizó el framework **express** para desarrollo de APIs y sitios web. Se optó por arquitectura REST, la cual se basa en HTTP y utiliza métodos CRUD (*Create, Read, Update, Delete*) para interactuar con la información de la base de datos.

## 7.2. Desarrollo de la API

Si bien la aplicación web no cuenta con ninguna interfaz gráfica, se la desarrolla según la arquitectura MVC (Modelo, vista, controlador). En la figura 24 se observa la organización de carpetas de la API.

```
.  
└── app/  
    ├── config/  
    │   ├── dev.js  
    │   ├── prod.js  
    │   └── index.js  
    ├── controllers/  
    │   └── package.controller.js  
    ├── middlewares/  
    │   ├── error.handler.js  
    │   └── validation.handler.js  
    ├── routes/  
    │   ├── package.routes.js  
    │   └── index.js  
    ├── sequelize/  
    │   ├── config/  
    │   ├── models/  
    │   └── sequelize.js  
    ├── services/  
    │   └── package.service.js  
    ├── validation/  
    │   └── createPackage.schema.js  
    ├── index.js  
    ├── package.json  
    ├── .sequelerc  
    └── .env
```

Figura 24: Árbol de organización de archivos de API.

A continuación se lista la funcionalidad de los archivos contenidos en cada carpeta:

- **config**: contiene los archivos de configuración con las credenciales y variables de entorno necesarias.
- **controllers**: gestiona la interacción entre la información que llega hacia una ruta y los servicios (o *services*)
- **routes**: declara las rutas de la API y las relaciona con un controlador específico.
- **services**: gestiona el flujo de información entre la API y la base de datos.
- **sequelize**: define y configura el ORM utilizado para el manejo de las funciones de la base de datos desde la API.

- **validation**: define los esquemas de validación necesarios para que la información que se guarde en la BD sea correcta.
- **middlewares**: Define los middlewares de validación y control de errores.
- **index.js**: Archivo principal de ejecución de la API.
- **.sequelizerc**: Define las carpetas de configuración del paquete ORM *sequelize*.

### 7.3. Definición de rutas

Se desarrollaron 3 rutas principales: una de ellas destinada a interactuar con la aplicación y las otras 2 para interactuar con el sistema de control de stock de la empresa.

Solicitud	Ruta	Uso
GET	<code>http://&lt;host&gt;/api/v1/</code>	Utilizada por la empresa para obtener la información de todos los paquetes.
GET	<code>http://&lt;host&gt;/api/v1/&lt;numero_de_paquete&gt;</code>	Utilizada por la empresa para obtener la información de un paquete.
POST	<code>http://&lt;host&gt;/api/v1/</code>	Utilizada por la Aplicación del sistema embbebido para publicar los datos en la BD.

Cuadro 3: Cantidad de imágenes de entrenamiento

### 7.4. Configuración de Máquina virtual en Azure

Para poder desplegar la API, se creó y configuró una máquina virtual en la nube de Microsoft, denominada *Azure Cloud*. Una vez creada la VM, se procedió a la apertura de los puertos necesarios y mostrados en la tabla 4.

Puerto	Tipo	Uso
22	Entrada	SSH para conexión remota, despliegue y configuración.
3000	Entrada	Puerto de escucha de la API.
3306	Salida	Comunicación con la BD remota.

Cuadro 4: Puertos configurados en la máquina virtual.

### 7.5. Diagrama de secuencia.

Para una mejor comprensión del funcionamiento interno de la API es necesario complementar lo explicado previamente con un diagrama de secuencias. Esto permite observar la

interacción entre los diferentes objetos y módulos que componen la arquitectura desarrollada. Para ello, se definieron 2 casos de uso sobre los cuales se basan los diagramas de secuencias desarrollados:

- Caso de Uso 1: Un operario de control de stock de la empresa desea obtener la información de un paquete con un número específico conocido y previamente cargado en la base de datos. El diagrama de secuencia para este caso se aprecia en la figura 25.
- Caso de Uso 2: La aplicación realiza una solicitud para crear un paquete nuevo en la base de datos. Los datos del paquete son validados correctamente, y la tarea se ejecuta sin errores. Finalmente, la aplicación recibe un código HTTP indicando que el paquete pudo ser añadido a la base de datos. El diagrama de secuencia para este caso se aprecia en la figura 26.

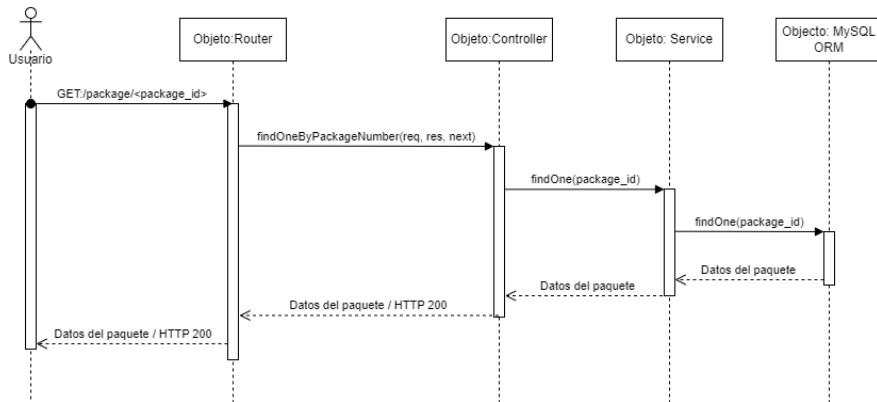


Figura 25: Diagrama de secuencia de API para primer caso de uso.

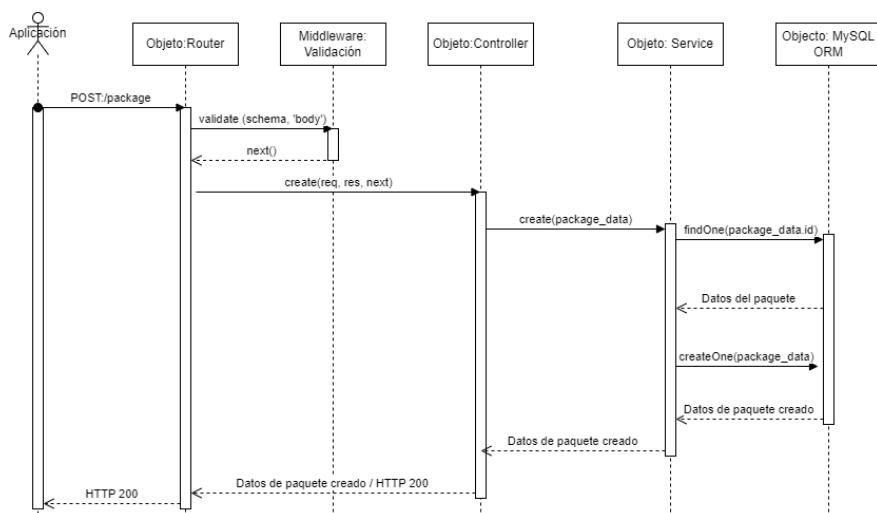


Figura 26: Diagrama de secuencia de API para segundo caso de uso.

## 7.6. Despliegue y ejecución de la API.

Una vez terminado el desarrollo de la aplicación, se clonó el repositorio con el código creado en la máquina virtual desplegada y se ejecutó mediante el comando **npm run start** para inicializar la API. En la figura 27 se puede ver que la API queda a la escucha de solicitudes en el puerto 3000.

```
azureuser_rpi@RpIAPIvm:~/app/RPiDetectServer/app/sequelize$ npm run start
> rpidetectserver@1.0.0 start
> node index.js

[2023-10-03T22:35:16.392] [INFO] sqlite-config-service - Database name: sql10644878
[2023-10-03T22:35:16.394] [INFO] sqlite-config-service - Package table name: package
Server listening to port: 3000
```

Figura 27: Salida por pantalla una vez ejecutada la API.

Para realizar pruebas se utilizó la aplicación de **Postman**, la cual permite enviar solicitudes a APIs bajo diferentes protocolos y formatos. En la figura 28 se puede apreciar un ejemplo de solicitud POST de un paquete y su correspondiente respuesta.

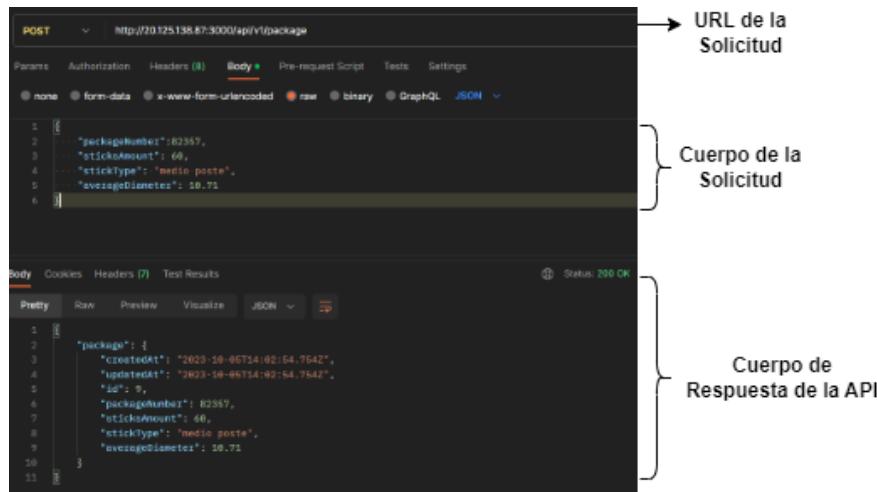


Figura 28: Captura de pantalla luego de una solicitud tipo POST de un paquete.

Por otro lado, si se quiere acceder a los datos almacenados en la base de datos de los paquetes, se puede realizar una solicitud del tipo GET, indicando el número de paquete a obtener. Un ejemplo de esto se muestra en la figura 29.

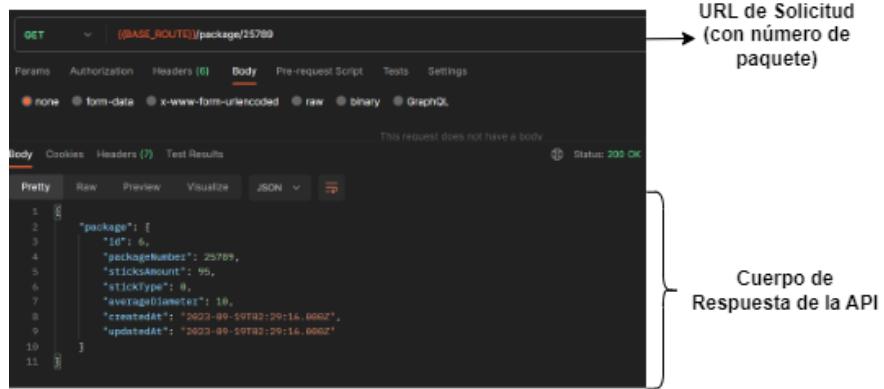


Figura 29: Captura de pantalla luego de una solicitud tipos GET de para obtener los datos de un paquete.

## 8. Parte 3: Aplicación de Integración

El objetivo de la aplicación es ejecutar el proceso de detección, calcular el diámetro promedio de palos detectados, mostrar el resultado al operario, enviar la información obtenida al servidor y también almacenarla de forma local. Todo esto mediante una interfaz gráfica adecuada para que el operario pueda realizar las tareas de forma precisa y cómoda.

La aplicación fue desarrollada con el framework **PyQt** versión 5, que presenta múltiples ventajas a la hora de combinar interfaces gráficas con tareas complejas de procesamiento. Además, es posible ejecutarlo en una SBC sin problema alguno.

### 8.1. Interfaz

Para crear la interfaz gráfica de la aplicación, se empleó una herramienta del framework Qt. La misma fue diseñada para el desarrollo de interfaces gráficas y es conocida como **Qt Designer**. Esta herramienta permite la creación visual de interfaces, que posteriormente se traducen al lenguaje de programación adecuado para la aplicación. En este caso, se utiliza python,

En la imagen 30 se puede ver la interfaz de usuario de la aplicación.

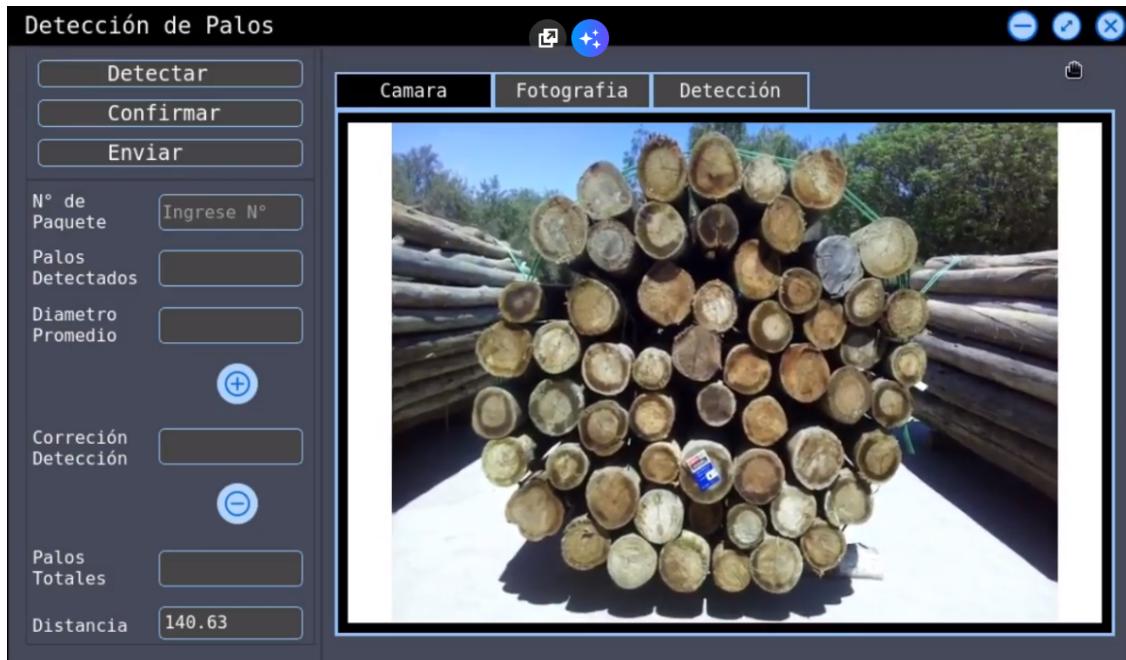


Figura 30: Interfaz de usuario.

La interfaz tiene presente tres botones en la sección superior izquierda:

- **Detectar:** Al presionar el botón, se ejecuta el algoritmo de detección. Luego, se muestran los resultados de **cantidad de palos detectados** y **diámetro promedio**.
- **Confirmar:** Al seleccionar este botón se confirma que los datos mostrados en pantalla sean correctos, de esta manera se asegura que la información subida al servidor sea verificada.
- **Enviar:** Al presionar este botón, se envían los datos al servidor. Esto sucede, siempre y cuando, haya algún resultado en la detección, se haya ingresado el número de paquete y el botón de confirmación se haya presionado. Además, este botón también ejecuta el guardado de los datos en el sistema de archivos local de la SBC.

Por otro lado, debajo de la sección de los botones se muestra lo siguiente:

- **Nº de Paquete:** El usuario debe ingresar el número de paquete correspondiente luego de la detección. De esta manera, se puede identificar el fardo en el servidor.
- **Palos Detectados:** Muestra la cantidad de palos que detectó el algoritmo.
- **Diámetro Promedio:** Muestra el promedio de los diámetros de los palos detectados.
- **Corrección Detección:** En caso de que el operario considere que hay errores en la detección (detecciones de palos extra, o faltantes), puede corregir agregando o disminuyendo palos con los botones de suma (+) y resta (-) respectivamente.

- **Palos totales:** Son los palos resultantes luego de aplicar la corrección, en caso de que no haya nada que corregir los palos totales serán los mismos que los palos detectados.
- **Distancia:** Muestra la distancia de la cámara al fardo medida por un sensor de proximidad ultrasónico.

Por último, en la parte de la derecha se pueden observar tres pestañas que muestran diferentes imágenes:

- **Cámara:** Muestra aquello captado por la cámara en tiempo real.
- **Fotografía:** Presenta la fotografía del fardo tomada al momento de detección, sin ningún procesamiento de la imagen.
- **Detección:** Expone los palos detectados en la fotografía del fardo tomada por la cámara al momento de detectar. Los palos detectados tienen un círculo amarillo para poder ser diferenciados fácilmente.

## 8.2. Desarrollo de la App

Para la implementación de la aplicación se utilizó un enfoque multi hilo (*multi-thread*), para evitar que las diferentes operaciones de procesamiento complejo (detección, solicitudes al servidor y medición de distancia) bloqueen la interfaz gráfica. Para comprender mejor el flujo de los diferentes hilos, en la figura 31 se observa un diagrama de secuencia que solo muestra la interacción entre el hilo principal y los 3 hilos secundarios o *worker threads*. Es decir, no se observa la interacción con el resto de las instancias presentes en la aplicación. Todas las instancias de hilos secundarios son creadas una vez iniciado el hilo principal, pero son ejecutados conforme el operario va realizando las tareas correspondientes.

El caso de uso presentado en el diagrama corresponde con un operario que realiza tareas en el siguiente orden:

- Inicialización de la aplicación
- Presionado del botón detectar, para ejecutar el algoritmo de detección.
- Una vez finalizado el algoritmo de detección, ingresa el número de paquete.
- Presiona el botón confirmar.
- Envía los datos al servidor con éxito.

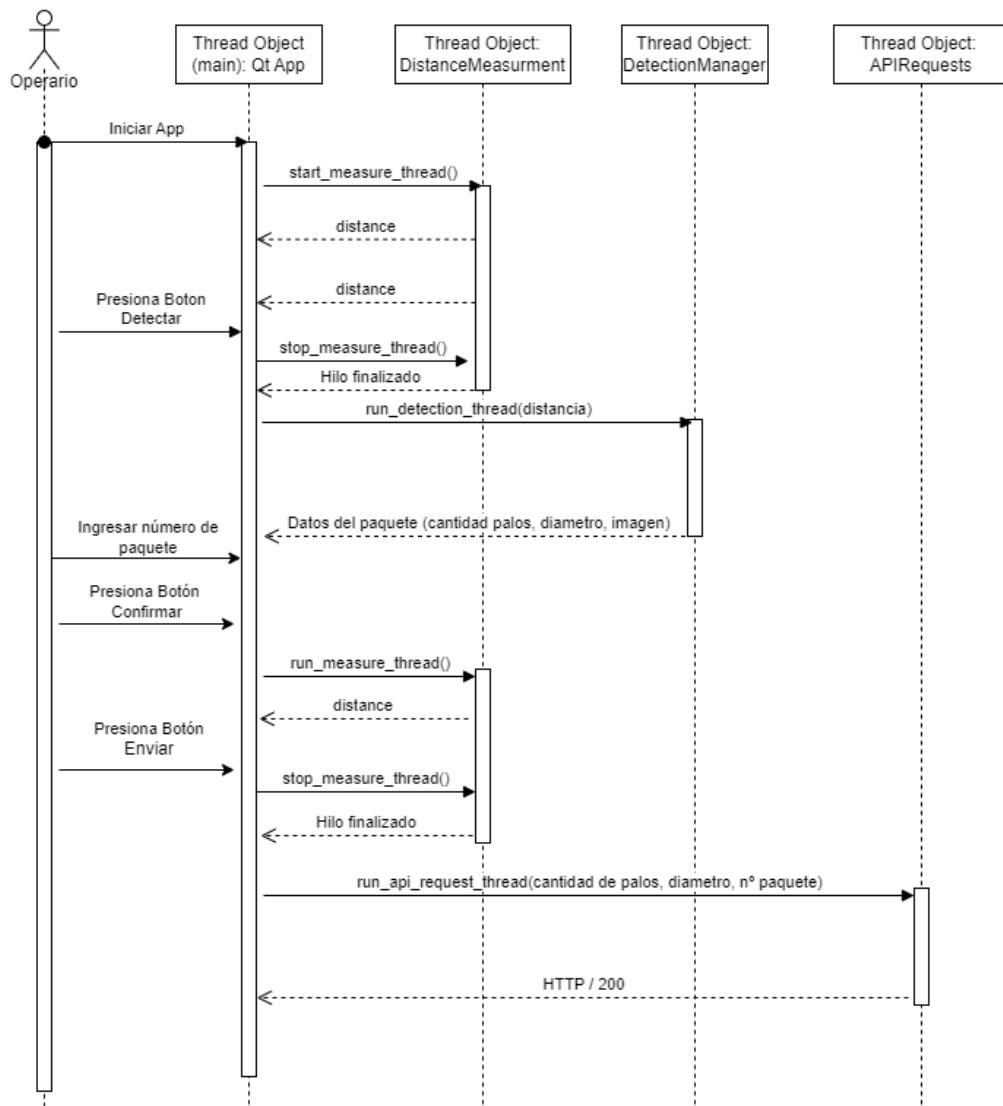


Figura 31: Diagrama de secuencia que muestra interacción entre diferentes hilos de la aplicación.

Finalmente, en la figura 32 se observa la organización de carpetas de la aplicación.

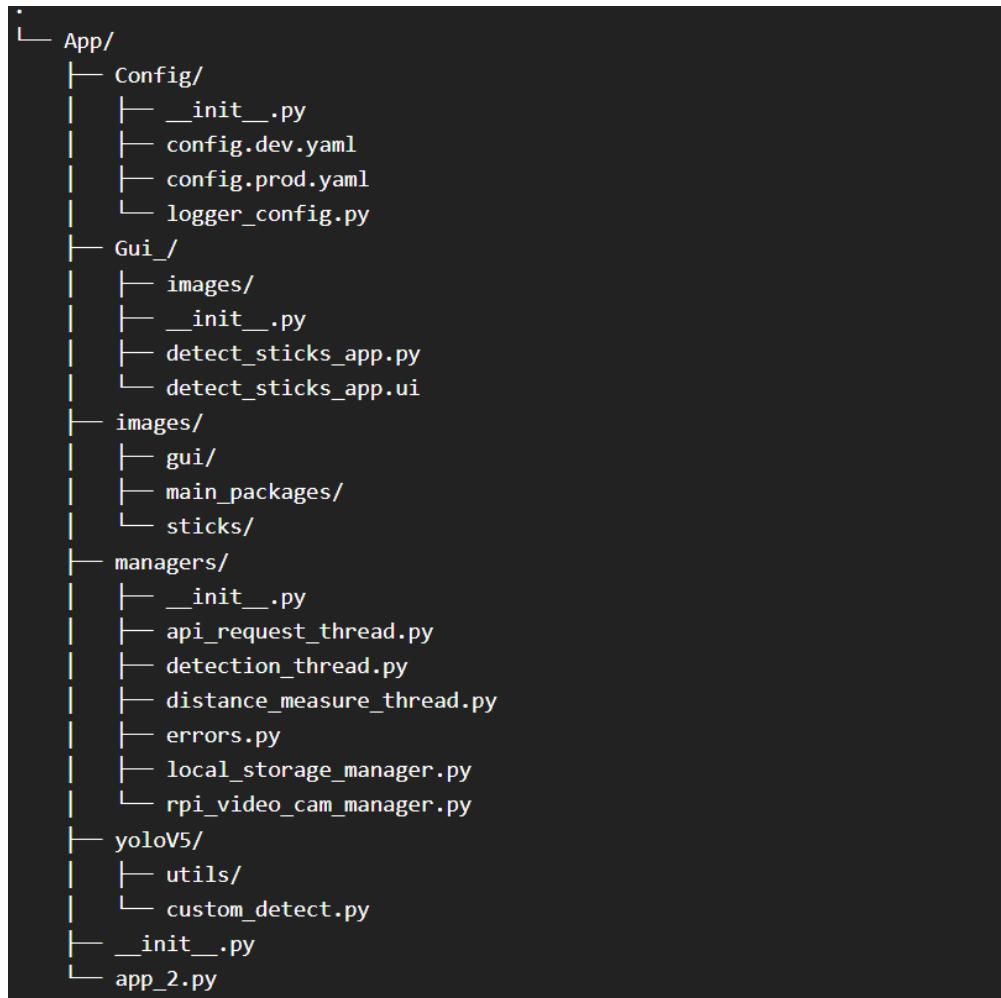


Figura 32: Árbol de organización de archivos de la aplicación.

A continuación se lista la funcionalidad de los archivos contenidos en cada carpeta:

- **config**: Contiene los archivos de configuración necesarios de la aplicación y del algoritmo de detección específicamente.
- **Gui**: Define y configura la interfaz gráfica.
- **images**: Contiene las carpetas donde se guardan las imágenes, y las imágenes que utiliza la interfaz gráfica.
- **managers**: Contiene archivos que funcionan de forma independiente entre sí, y que son utilizados por la aplicación principal. Entre otros archivos, contiene la definición de los hilos secundarios.
- **yoloV5**: Contiene el algoritmo de detección, cuyo archivo principal (**custom\_detect.py**) es ejecutado desde el hilo secundario ubicado en **managers/detection\_thread.py**.

- **app.py**: Archivo principal de ejecución de la aplicación. Contiene la implementación del hilo principal.

### 8.3. Cálculo del diámetro promedio

Al finalizar el proceso de detección, se obtiene el diámetro promedio de los palos que componen del paquete. Para implementar dicho cálculo, primero se obtuvo la imagen de un objeto conocido (a una distancia determinada de la cámara) y se midió su tamaño en la realidad. Luego, se midió su tamaño en píxeles y, con dicha información, se pudo obtener una calibración inicial que considera la relación de píxeles a centímetros para la cámara utilizada y la distancia al objeto considerada. De esta manera, el cálculo mostrado en la ecuación 5 se realiza solo una vez para la calibración.

$$relacion_{cm-px-calibracion} = \frac{tamaño_{objeto-real}(cm)}{tamaño_{objeto-imagen}(pixeles)} \quad (5)$$

Una vez obtenida la calibración inicial, se escaló dicha relación de píxeles a centímetros inicial para cualquier otra distancia de la cámara al objeto. Específicamente, para cualquier otra distancia entre la cámara y el paquete de palos. Para este caso, la distancia mencionada es medida mediante el uso de un sensor ultrasónico **HC-SR04**. De esta manera, para obtener la escala de centímetros a píxeles para una distancia determinada del paquete:

$$relacion_{cm-px} = \frac{relacion_{cm-a-px-calibracion} * distancia_{medida}}{distancia_{calibracion-original}} \quad (6)$$

Finalmente, al obtener el diámetro en píxeles como salida del algoritmo de detección, se obtiene el diámetro real de un palo en centímetros como se muestra en la ecuación 7.

$$diametro(cm) = relacion_{cm-px} * diametro(pixeles) \quad (7)$$

La información de calibración original se encuentra en el archivo de configuración de la aplicación. De esta manera, se busca evitar cambios en el código en caso de necesitarse una nueva calibración.

### 8.4. Hardware utilizado

Se realizaron las conexiones del hardware mostrado en la figura 33 siguiendo el siguiente procedimiento:

1. La pantalla LCD touch screen de 7 pulgadas se conecta a la Raspberry Pi 4 mediante un cable HDMI para la transmisión de la imagen y un cable USB para suministrar corriente.

2. La Raspberry Pi Camera V2.1 se conecta a la Raspberry Pi 4 mediante un cable flex display.
3. El sensor de proximidad ultrasónico HC-SR04 se conecta a la Raspberry Pi 4 utilizando cables tipo Dupont, los cuales se conectan a los pines correspondientes en la Raspberry Pi. Además, se emplean resistencias para limitar la tensión a la que se someten los pines de la raspberry, ya que el sensor genera señales de 5V y los pines soportan hasta 3.3V.

Para visualizar de manera gráfica estas conexiones, se adjunta el esquema en la imagen 33 del informe.

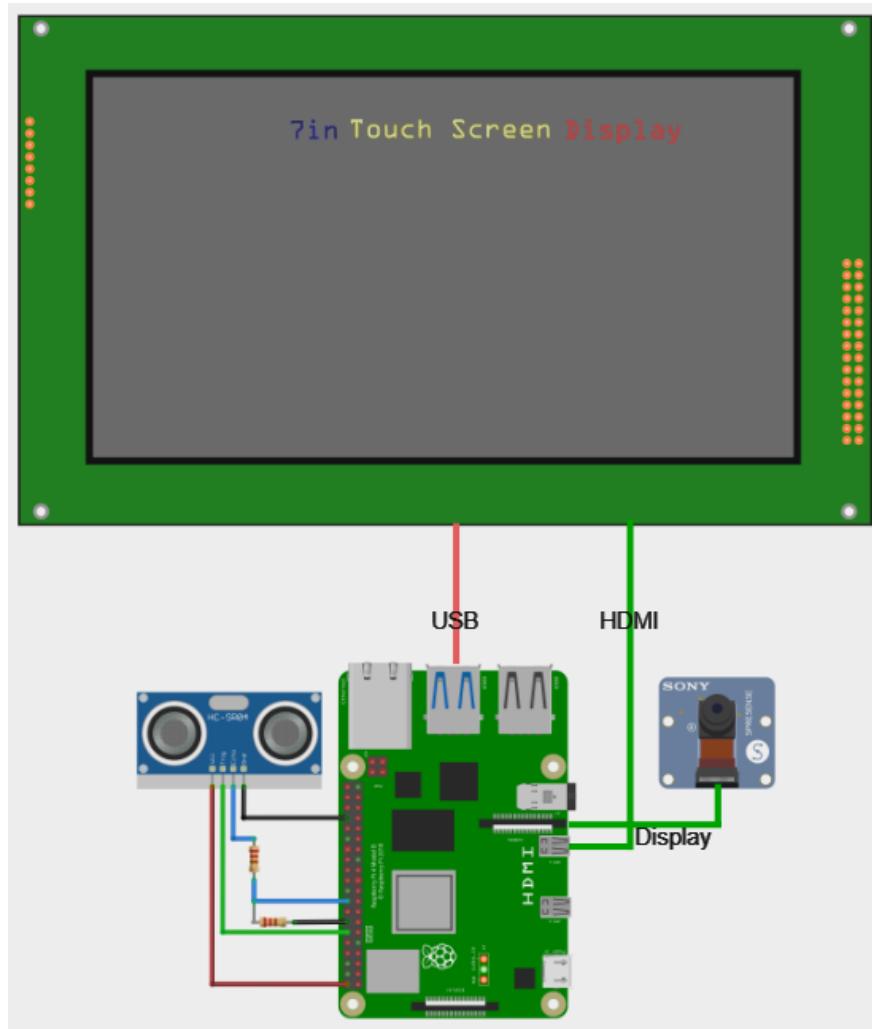


Figura 33: Conexiones de la Aplicación.

#### 8.4.1. Sensor de distancia HC-SR04

El sensor ultrasónico HC-SR04 se basa en el principio de medición del tiempo de vuelo para determinar la distancia entre el sensor y un objeto (en este caso, el paquete de palos).

Emitiendo pulsos ultrasónicos de alta frecuencia (343 m/s) y registrando el tiempo que tarda en recibir el eco del objeto, calcula la distancia con una resolución de **0.3cm**. Su rango de medición es de **2 hasta 400 cm**. Este sensor cumple con los requerimientos de precisión y, además, su rango es adecuado ya que para este caso de uso, la cámara y el sensor no se ubicarán a mas de 2 metros del paquete a detectar.

Para su manipulación en el código, se utilizó la librería **GPIO** de python. La misma permite el acceso a los pines de la Raspberry, tanto para su lectura y escritura. A continuación se muestra una tabla con los pines utilizados y su respectiva conexión en el sensor.

GPIO Pin	Pin Sensor
18	Trigger
24	Echo
5V	VCC
GND	GND

Cuadro 5: Detalle de conexiones Sensor HCSR04.

#### 8.4.2. Cámara

La cámara utilizada se denomina *Raspberry Pi Camera V2.1*, especialmente diseñada para ser utilizada por la raspberry pi. Cuenta con una resolución de 8 megapíxeles y es capaz capturar imágenes en 3280 x 2464 píxeles. Esto es suficiente para incorporar la imagen en la red neuronal sin perder relación de aspecto, ya que YOLOv5 acepta entradas en diferentes resoluciones. Para la manipulación en el código, se utilizó la librería de python denominada **picam2**, diseñada para administrar las diferentes funcionalidades sobre esta cámara. Por otro lado, la conexión desde la cámara hacia la raspberry pi se realizó mediante un cable *flex display*.

#### 8.4.3. Pantalla Táctil LCD de 7 pulgadas

La pantalla utilizada es útil para mostrar la interfaz de usuario y manipular la aplicación. La misma se conectó mediante un cable USB , que le brinda energía, y un cable HDMI que proporciona la imagen a mostrar.

### 8.5. Procedimiento de uso de la aplicación

El proceso comienza con el operario ingresando a la aplicación. Una vez dentro, el usuario se encuentra en la pantalla de inicio, que fue previamente descrita en la sección de la interfaz de usuario.

A continuación, el operario debe asegurarse de que la cámara esté correctamente enfocada hacia el paquete, controlando la altura de la para evitar perder algún palo durante la detección. Después, se debe presionar el botón 'Detectar' para iniciar el proceso de detección y esperar a que este finalice para visualizar los resultados.

Luego, se le solicita al operario que ingrese el número de identificación del fardo, que es de utilidad para ser reconocido en la base de datos. Posteriormente, se procede a la revisión de los datos resultantes de la detección, lo que implica la revisión de la imagen y la corrección, en caso de ser necesaria. Una vez verificados los datos, se presiona en el botón 'Confirmar'.

Finalmente, para cargar los datos en el servidor, se debe pulsar botón 'Enviar'. Si todas las etapas anteriores se completan con éxito, los datos se envían al servidor y se mostrará un mensaje de éxito. En caso contrario, se mostrará un mensaje de error y se reiniciará el proceso. Además, ya sea que los datos se guardan correctamente o no en el servidor, siempre son guardados de forma local en la SBC.

El siguiente diagrama de flujo proporciona una representación gráfica de estos pasos.

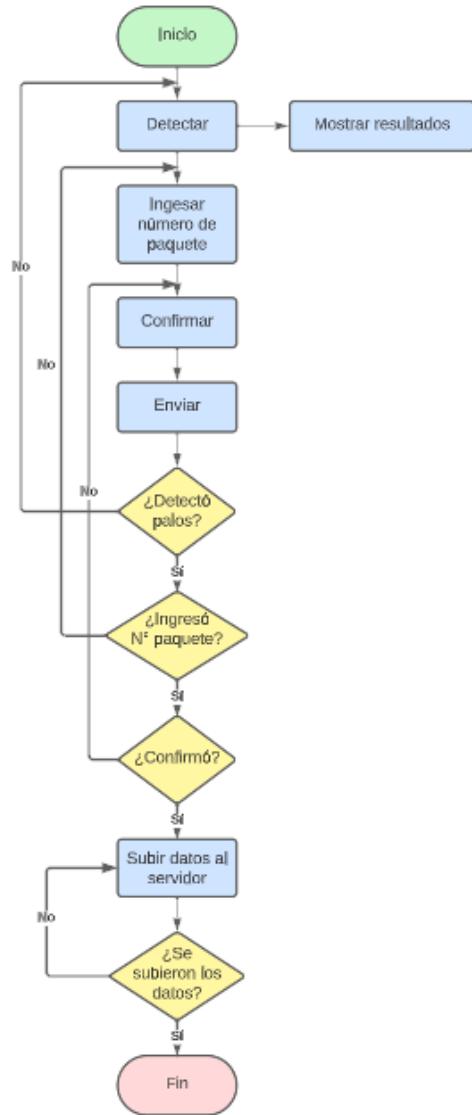


Figura 34: Diagrama de flujo de la Aplicación.

## 9. Resultados

En este caso se utilizó la aplicación para detectar los palos de un paquete de 52 unidades. A continuación, se presentan los resultados de esta prueba junto con capturas de pantalla de la aplicación.

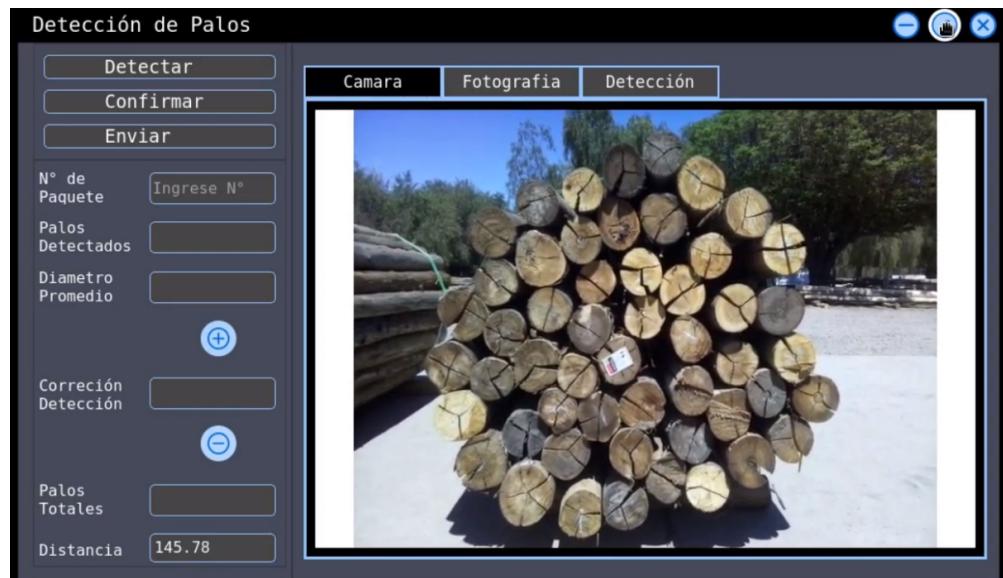


Figura 35: Vista de Cámara en la aplicación antes de detectar.

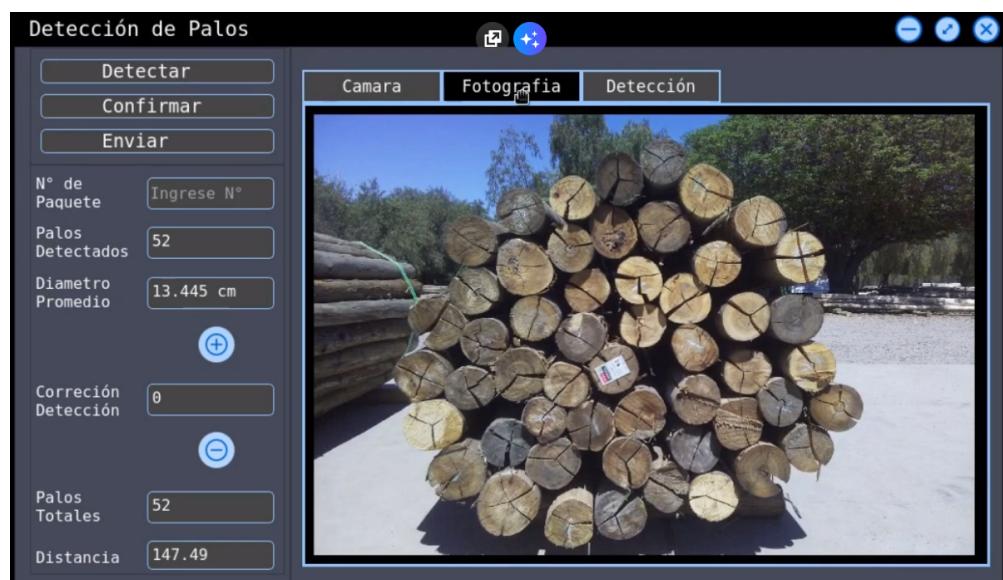


Figura 36: Vista de Fotografía en la aplicación.



Figura 37: Vista de Detección en la aplicación.

Las capturas de pantalla de la aplicación muestran de manera concluyente que la detección de palos fue precisa, logrando contar correctamente los 52 palos presentes en el paquete. En este caso, se encontró que el diámetro promedio estimado por la aplicación fue de 13,445 cm, lo que coincide con el rango de 12 a 14 cm de diámetro esperado para este tipo de paquete.

Una vez que se corroboró la información de la detección y se ingresó el número de paquete correspondiente, se procedió a confirmar y enviar los datos al servidor. Los datos se almacenaron correctamente, como se observa en el mensaje de éxito.



Figura 38: Mensaje de subida de datos al servidor exitosa.

En caso de que surja algún error al enviar los datos al servidor, como la inserción de un número de paquete que ya está en uso por un paquete anterior, se mostrará un mensaje de advertencia. El operario tendrá la opción de corregir el número de paquete y reintentar el envío de los datos.



Figura 39: Advertencia de falla en subida de datos al servidor.

En el segundo escenario, se utilizó la aplicación para detectar palos en un paquete que contenía un total de 59 unidades, con un rango de diámetro estimado de 10 a 13 cm. A continuación, se presentan los resultados de esta prueba junto con las capturas de pantalla de la aplicación.

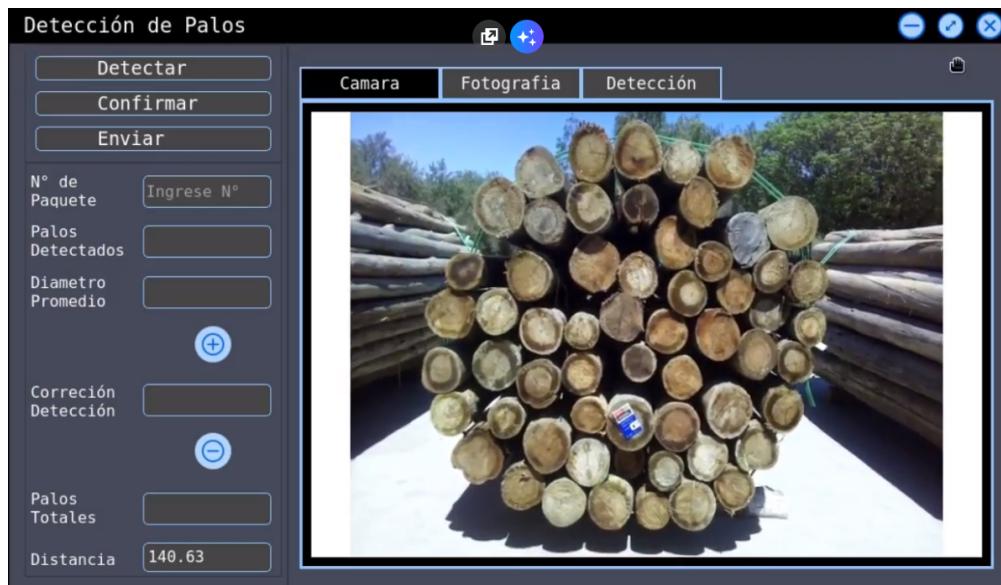


Figura 40: Vista de Cámara en la aplicación antes de detectar.



Figura 41: Vista de Fotografía en la aplicación.

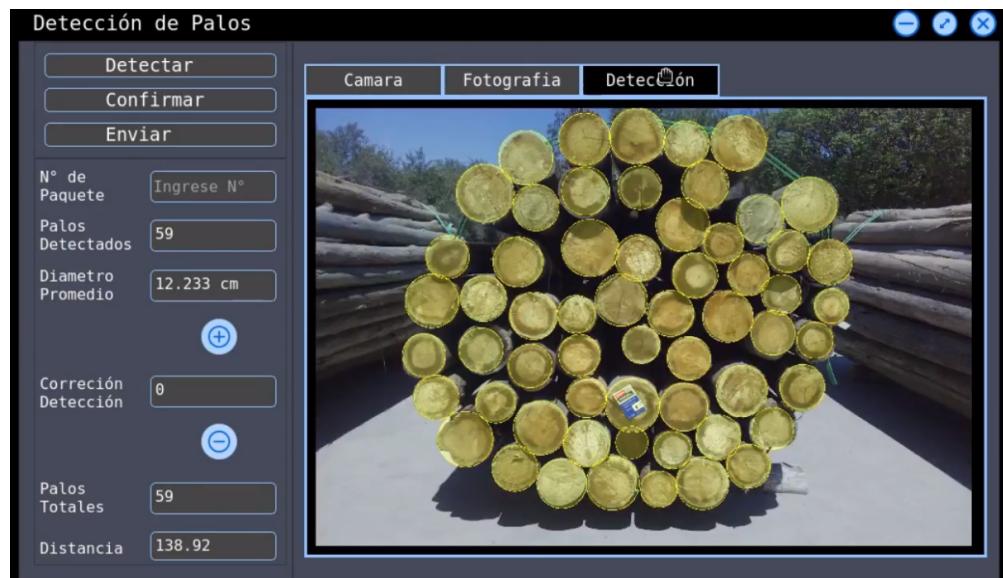


Figura 42: Vista de Detección en la aplicación.

Los resultados de este caso también resultaron satisfactorios, al detectar correctamente los 59 palos del paquete y 12,233cm de diámetro promedio. Este valor se ajusta adecuadamente al rango de diámetro de este tipo de paquete, que se considera entre 10 y 13 cm.

Con estos resultados satisfactorios, se concluye que la aplicación cumple con los objetivos propuestos en un principio.

## **10. Conclusiones**

Durante el transcurso de este proyecto se logró optimizar el proceso productivo de una industria dedicada al rubro de la madera, principalmente para su uso en vitivinicultura. Específicamente, se agilizó el proceso de conteo de troncos mediante el uso de redes neuronales, reduciendo los tiempos de trabajo en un 99 % y perfeccionando la calidad de la tarea. Durante las pruebas e interacción con la empresa, se logró estimar un ahorro de tiempo de 2 meses de trabajo (anualmente) de un operario, además del perfeccionamiento a la hora de realizar la tarea. De esta manera, el operario antes encargado del conteo de palos, ahora puede utilizar ese tiempo en realizar otras tareas menos repetitivas, menos nocivas para su salud y mas efectivas para el negocio. Paralelamente, el uso de este algoritmo permitió obtener una estimación del diámetro promedio de un conjunto de troncos, lo cual es fundamental para las actividades de la empresa y que, actualmente, no esta siendo ejecutado. Además, se integraron los resultados obtenidos del proceso de detección con el sistema de control de stock de la empresa, permitiendo un acceso simplificado a los datos y mejorando el flujo de información dentro de la empresa. De esta manera, se consideran cumplidos los objetivos técnicos planteados en un principio.

En otro orden, lo explicado anteriormente logró aplicarse y probarse en una empresa de la región mediante un primer prototipo, permitiendo la incorporación de nuevas tecnologías al campo productivo local. Esta implementación ha allanado el camino para que el sector maderero provincial se vuelva más tecnificado y competitivo, lo que, a su vez, contribuye al crecimiento económico y al fortalecimiento de la base industrial de la región.

## **11. Trabajos Futuros**

Hasta aquí finalizan las actividades y desarrollos planteados como proyecto final de carrera. Sin embargo, todavía existen mejoras a realizarse y complementos a agregar. Este capítulo sirve para mencionar futuras características que se deberían adicionar para que, finalmente, el desarrollo sea utilizado en el ambiente productivo:

- Reemplazar la pantalla de 7 pulgadas por una de mayor tamaño, para una mayor facilidad de manipulación por parte del operario de campo.
- Adicionar control de acceso a la API, para protección de la información.
- Integración con la base de datos real de la empresa, ya que para este proyecto se utilizó una base de datos de prueba, no en el ambiente productivo real. Este proceso involucraría la creación de la tabla en el servidor de la empresa y la reconfiguración de permisos y rutas.
- El uso de una cámara con capacidad de medición de distancias, para mayor precisión en el cálculo del diámetro. Si bien la precisión de distancia es cumplida por el sensor real utilizado hasta el momento, se estima que todos los palos del paquete se encuentran a la misma distancia de la cámara. Si bien esto es aproximadamente real, pueden existir diferencias de hasta 1 cm, que pueden ser consideradas en caso de utilizar este tipo de cámara.

## 12. Referencias

Redmon, J., Divvala, S., Girshick, R., Farhadi, A. (2016). *You Only Look Once: Unified, Real-Time Object Detection*. IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Thuan, D. (2021). *Evolution of YOLO Algorithm and YOLOv5: The State-of-the-art Object Detection Algorithm* (Bachelor's thesis, Oulu University of Applied Sciences).

Li W, Cheng J, Chen B, Xue Y, Wang Y, Fu Y, et al. (2023). *MaskID: An effective deep learning-based algorithm for dense rebar counting*. PLoS ONE 18(1): e0271051. <https://doi.org/10.1371/journal.pone.0271051>

Modelo de ultralytics yolov5: [Link Modelo YoloV5 ultralytics](#)

Generación de anchor boxes con k-means: [Link a generación de anchor boxes](#)

Cómo entrenar un modelo propio de YoloV5: [Link How to train a custom yolov5 model.](#)

Documentación de librería pytorch: [Link Pytorch documentation](#)

Documentación de Qt Designer: [Link Documentación de Qt Designer.](#)

Documentación de sensor HCSR04: [Link Datasheet sensor HCSR04](#)

Documentación de cámara Raspberry Pi V2.1: [Link Documentación de cámara raspberrypi](#)

Documentación sobre Qt threads: [Link Documentación Qt threads](#)

Sitio web de Count things app: <https://countthings.com/>

## **13. Anexos**

- Link de código aplicación embebida: <https://github.com/MARIANOCEREDA/RaspberryPiDetect>
- Link de código API: <https://github.com/MARIANOCEREDA/RPiDetectServer>
- Link de videos de prueba: [https://drive.google.com/drive/folders/1KqaBZKMYRwfbl\\_NHLNDpkcdLwKMANh7J?usp=drive\\_link](https://drive.google.com/drive/folders/1KqaBZKMYRwfbl_NHLNDpkcdLwKMANh7J?usp=drive_link)
- Link a ejemplos de detecciones: <https://drive.google.com/drive/folders/detecciones>

Archivo de configuración de la Aplicación:

```
Archivo de configuración de la App (config.prod.yaml)

environment: prod
local_storage_folder: <App folder>/images

camera:
    res_x: 3000
    res_y: 2000
    w_obj_ref_px: 100
    h_obj_ref_px: 155
    w_obj_ref_cm: 1.8
    h_obj_ref_cm: 2.92
    w_obj_real_cm: 5.6
    h_obj_real_cm: 9.1
    distance_cm: 139

yolov5:
    save_txt: True
    line_thickness: 2
    hide_labels: True
    device: "cpu"
    img_size: 640
    source:  <App folder>/images
    img_size_w: 640
    img_size_h: 640

    images:
        results: <App folder>/images/main_packages

    sticks:
        weights:  <App folder>/yoloV5/weights/sticks/best.pt
        results: <App folder>/images/sticks
        conf_thres: 0.62
        iou_thres: 0.45
```

Archivo de ejecución de la Aplicación:

## Archivo de ejecución de la App (run.sh)

```
#!/bin/bash
DETECT_FOLDER="/home/mariano/workspace/tesis/RaspberryPiDetect"
APP_FOLDER="/home/mariano/workspace/tesis/RaspberryPiDetect/App"
LOG_FILE="/home/mariano/workspace/tesis/log/run.log"
echo "Script started at $(date)" | tee -a "$LOG_FILE" 2>&1
if [ ! -d "$DETECT_FOLDER" ]; then
    echo "Error: Detect folder not found." >> "$LOG_FILE"
    exit 1
else
    echo "Detect folder found."
fi
source "$DETECT_FOLDER/bin/activate" >> "$LOG_FILE" 2>&1
if [ $? -ne 0 ]; then
    echo "Error: Virtual environment activation failed." >> "$LOG_FILE"
    exit 1
else
    echo "Virtual env activated."
fi
cd "$APP_FOLDER" >> "$LOG_FILE" 2>&1
if [ $? -ne 0 ]; then
    echo "Error: App folder not found." >> "$LOG_FILE"
    exit 1
else
    echo "App folder found."
fi
echo "Executing app...
python app_2.py >> "$LOG_FILE"
if [ $? -ne 0 ]; then
    echo "Error: App execution failed." >> "$LOG_FILE"
    exit 1
else
    echo "App execution OK."
fi
deactivate
echo "Script completed at $(date)" | tee -a "$LOG_FILE" 2>&1
```