

# Ajuste fino y despliegue de un modelo YOLO 11n para detección de objetos con una Raspberry Pi 4B

José Ángel Amaro Blanco

11 de diciembre, 2025

*Disclaimer: Este reporte describe la etapa de un proyecto semestral de la materia de Visión Computacional impartida en la UDEM durante el semestre de Otoño 2025 por el Dr. Fermín Castro Aragón. El proyecto completo fue resultado del trabajo propio así como de otros 3 integrantes del equipo: Alejandro Lozano Vessi, Noé Garza Santos, y Andrés Mendoza. Por tanto, este reporte documenta la etapa mencionada del proyecto, la cual supervisé y desarrollé principalmente, excepto cuando se indique lo contrario. Un reporte similar fue entregado como trabajo del curso, y muchos contenidos se encuentran ahí también, entonces me veo en la necesidad de citarlo aquí [1].*

## 1. Introducción

En la etapa final del curso se nos pidió detectar 3 objetos utilizando el algoritmo de You Only Look Once (YOLO) en un cuarto siguiendo una rutina estilo *roomba*. Estos fueron los objetos:

- Un dinosaurio impreso en 3D.
- Un eje coordenado.
- Un haptic paddle.



Figura 1: Dinosaurio impreso en 3D

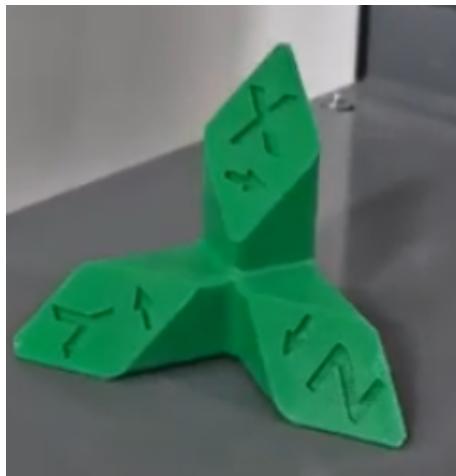


Figura 2: Eje coordenado



Figura 3: Haptic paddle

Su implementación consistió en dos etapas:

1. Ajuste fino de alguna versión de YOLO para detectar estos tres objetos
2. Implementar el algoritmo en un carro robótico controlado con una Raspberry Pi (RPi) y poder obtener detecciones a más de 12 frames por segundo (FPS).

## 2. Metodología

### 2.1. Ajuste fino de modelo YOLO

#### 2.1.1. Conjunto de datos

Para realizar el ajuste fino se tuvo que crear un conjunto de datos personalizado. Para la creación del conjunto de datos, integrantes del equipo:

1. Grabaron videos de las 3 piezas en distintos lugares e iluminaciones.
2. Segmentaron los videos de tal modo que tuviéramos 150 frames de cada uno con un script de Python.

Eventualmente terminamos con 150 imágenes para cada objeto. Estas luego se subieron a Roboflow en proyectos independientes (porque la idea era repartir la carga de labelling entre integrantes del equipo), y Alejandro dibujó las bounding boxes en el asistente de labelling de Roboflow.

Inicialmente terminamos con 300 imágenes de entrenamiento, 75 de validación, y 75 de prueba. Sin embargo, al entrenar modelos en la etapa inicial de este trabajo, nos dimos cuenta cualitativamente que al girar las piezas el modelo no las detectaba. En vista de esto, y a fin de poder validar el modelo con un conjunto de prueba más grande, distribuimos el conjunto de datos en 100 entrenamiento, 150 validación, y 200 prueba, para posteriormente hacer un preprocesamiento de datos (auto-orient y resize a 640x640), y luego hacer un aumento de datos en Roboflow solamente para los datos de entrenamiento. Realizamos distintos aumentos de datos y pruebas (las cuales se mencionan en la siguiente sección), pero el aumento de datos final fue el siguiente:

- Girar imágenes sobre eje horizontal y vertical.
- Rotación  $\pm 90^\circ$
- Shear  $\pm 10^\circ$  Horizontal y  $\pm 10^\circ$  Vertical
- Convertir a escala de grises el 15 % de las imágenes totales
- Variar el Hue entre  $-15^\circ$  y  $+15^\circ$  del valor original
- Variar Brightness entre 0 % y +20 %
- Aplicar Blur hasta 2.5 px

De modo que nuestro conjunto de datos final fue de:

- 500 entrenamiento
- 150 validación
- 200 prueba

### **2.1.2. Entrenamiento**

Tras contar con el conjunto de datos anotado, se empezó a hacer ajuste fino del modelo YOLO v8n. Se escogió la versión nano porque necesitábamos un modelo ligero para poder hacer inferencia en tiempo real desde la RPi. Se hizo el ajuste fino desde la interfaz de línea de comandos (CLI) de Ultralytics. Se usaba mínimo preprocesamiento (orientación automática), y se hicieron pruebas usando imágenes de entrada de 320x320, 480x480, y 640x640. No se observaron buenos resultados.

Luego se fueron haciendo pruebas agregando cada vez más aumento de datos. Tras observar resultados pobres (entre 2 y 5 FPS) se empezó a probar exportando los modelos .pt a formatos .onnx, .ncnn, y .mnn, optimizándolos para inferencia en

CPU, pues la GPU de la RPi no está optimizada para aprendizaje profundo, pero incluso observamos resultados ligeramente peores.

Después cambiamos a la versión 11 de YOLO porque en teoría es la más optimizada y más rápida. Además, ahí ya hicimos el aumento definitivo de datos que se mencionó en la sección anterior. A partir de aquí se buscó usar la siguiente estrategia para cada modelo de:

1. Hacer ajuste fino a partir de YOLO 11n, dejándolo correr 50 épocas.
2. Hacer ajuste fino del modelo anterior, con menor learning rate, y dejándolo correr por más épocas (100).

A continuación se muestran los hiperparámetros relevantes declarados explícitamente desde la terminal del ambiente virtual de python:

- **task**: Tarea a realizar por el algoritmo. En este caso, `detect`.
- **mode**: Modo de operación. En este caso, `train`.
- **model**: Arquitectura del modelo a usar. En este caso, `yolov11n.pt` u otro modelo entrenado.
- **data**: Ruta al archivo `.yaml` que contiene las rutas a los conjuntos de datos de entrenamiento, validación, y prueba.
- **epochs**: Número de épocas a entrenar. En este caso, 50 o 100
- **patience**: Número de épocas sin mejora en validación antes de detener el entrenamiento. En este caso, 10.
- **imgsz**: Tamaño de las imágenes de entrada. En este caso, 640.
- **cls**: Lo pusimos como 1 para darle el máximo peso a la pérdida de clasificación.
- **dropout**: Porcentaje de neuronas dropout en capa(s) entrenada(s). En este caso, 0.5.
- **lr**: Learning rate inicial. En este caso,  $5 \times 10^{-4}$  (estando cerca del valor estándar de  $1 \times 10^{-3}$ ) para el primer ajuste fino, y  $5 \times 10^{-5}$  para el segundo.
- **cos\_lr**: Ajustar learning rate de forma cosenoidal. En este caso, `True`.

También ajustamos estos otros hiperparámetros que pueden ser útiles para la reproducibilidad del experimento, o para el desarrollo de otros experimentos:

- **deterministic**: Lo pusimos como `False` para mayor velocidad de inferencia
- **seed**: Lo pusimos como 0
- **plots**: Crear gráficas de desempeño. Lo pusimos como `True`
- **project**: Carpeta donde se guardan los resultados del entrenamiento.
- **name**: Usamos formatos como “v11n-480-50-aug-rgb-gray-reb-blur-” para distinguir entre versiones de modelos, img sizes, aumentos de datos, etc.

Los demás hiperparámetros (que son muchos) se dejaron en sus valores por defecto.

### 2.1.3. Métricas de desempeño

Estos fueron los resultados de la última sesión de entrenamiento del modelo YOLO 11n con imágenes de tamaño 640x640:

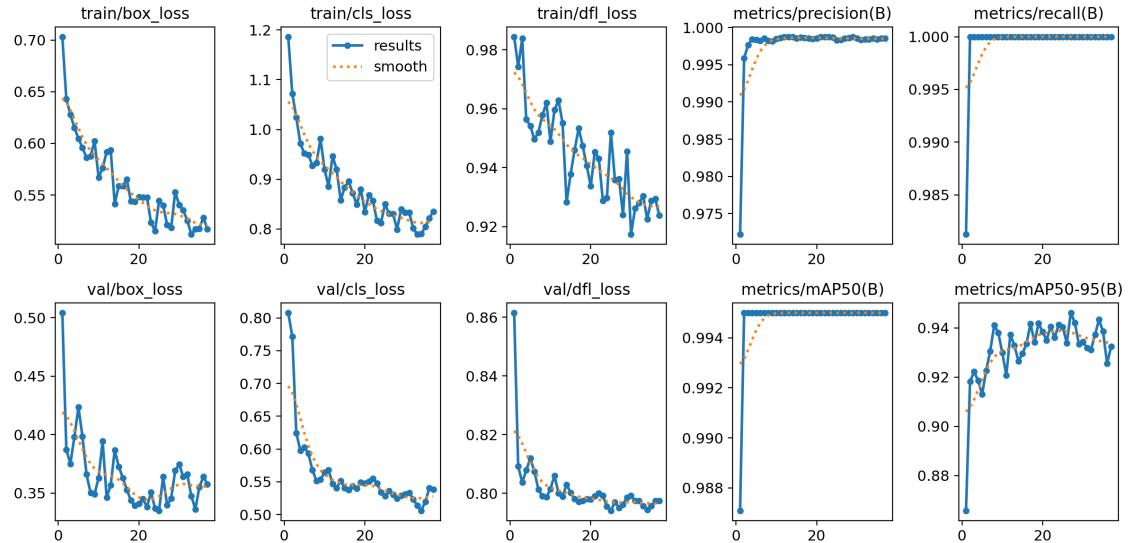


Figura 4: Resultados del entrenamiento del último modelo entrenado

En la siguiente figura se aprecia la máxima mAP50\_95 del modelo. Estaba configurado para guardar los mejores pesos, así que esa métrica corresponde al modelo final:

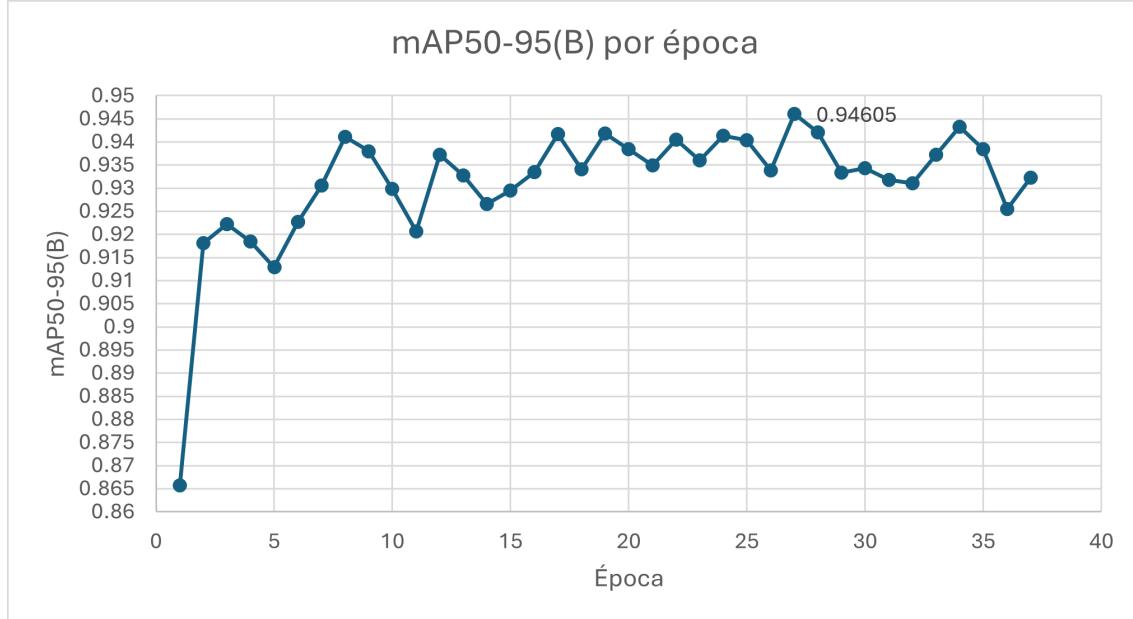


Figura 5: mAP50\_95 a lo largo de las épocas de entremamiento

Esta métrica indica, para distintos criterios de intersección sobre unión (IoU) para

hacer predicciones positivas, en promedio, la fracción del área compartida entre las cajas predichas y las cajas reales con respecto al área total combinada es de 94.6 %. Dicho de otro modo, las cajas predichas se "traslapan" mucho con las cajas reales.

También se obtuvo la siguiente gráfica f1-score con respecto al umbral de confianza para predicciones positivas del modelo:

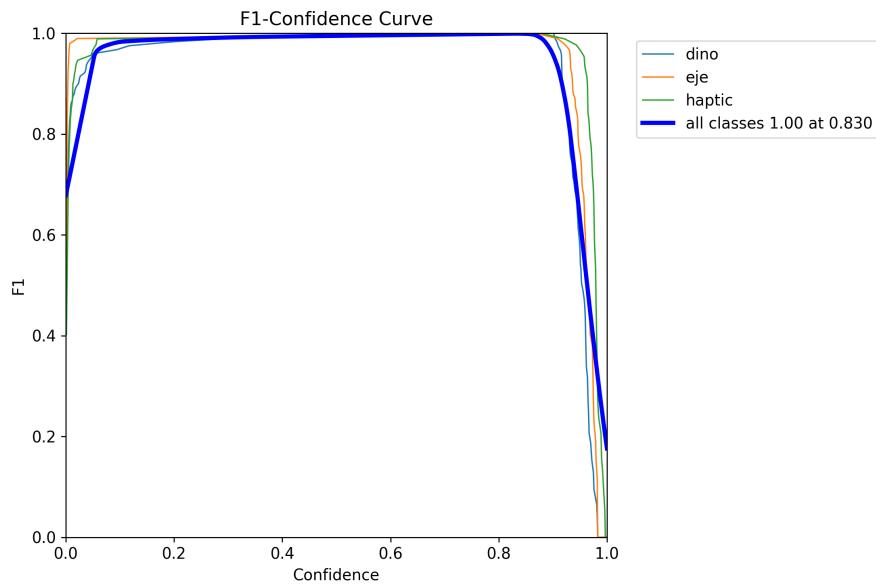


Figura 6: F1-score vs umbral de confianza

Con base en el mayor umbral de confianza que se tiene antes de observar una rápida caída en el f1-score (es decir, 83 %) se construye la siguiente matriz de confusión:

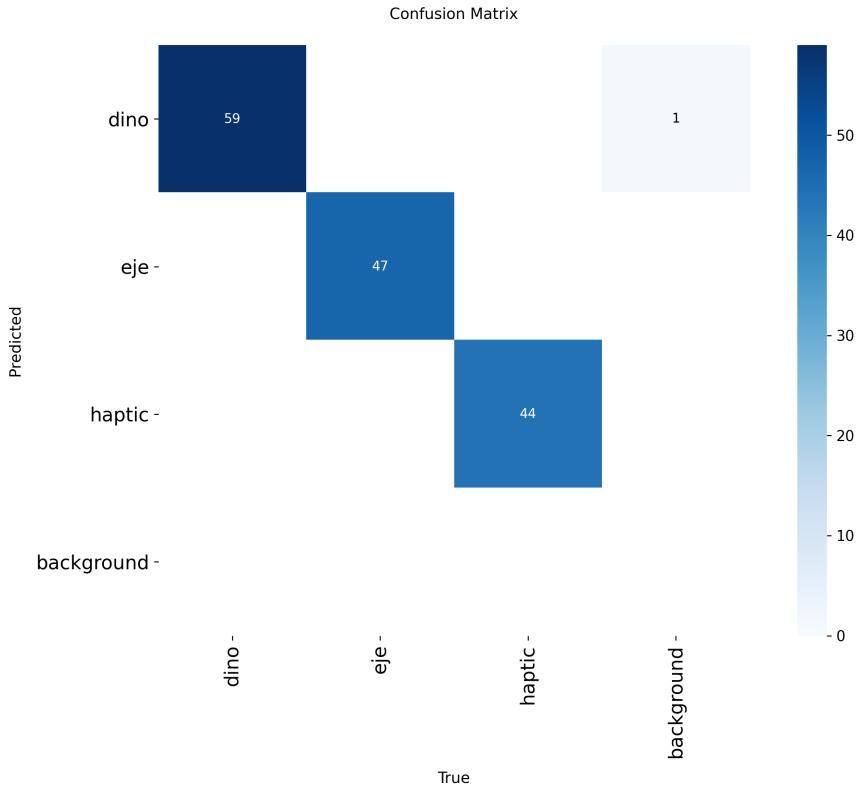


Figura 7: Matriz de confusión del modelo final con umbral de confianza del 83 %

Contando los 150 elementos de la diagonal principal en la Figura 7 y dividiéndolos entre los 151 en total podría llevarnos a pensar que el modelo tiene una accuracy de 99.3 %, pero la realidad es otra cuando consideramos que en realidad el conjunto de prueba consiste en 200 imágenes, y la matriz de confusión muestra un soporte de 151 imágenes. Esto quiere decir que de las 200 imágenes, solo 151 pasaron el umbral de confianza de 83 % del modelo para entrar dentro de las predicciones consideradas en la matriz de confusión. De modo que:  $150/200 = 0.75$ , y por tanto **en realidad el modelo tuvo una accuracy del 75 %**.

Sí cabe destacar que, experimentalmente, aun con el umbral de confianza de 83 % llegaban a ocurrir varios falsos positivos. Por tanto, si se baja el umbral de confianza para que más imágenes pasen el umbral de detección, probablemente haya más falsos positivos. En todo caso, el área de oportunidad recaería en incluir más imágenes en el dataset que no contengan ninguno de los objetos presentes; imágenes con archivos de anotación vacíos [2].

De todos modos, menciono que el mAP indica un muy buen desempeño del modelo, porque a diferencia de la accuracy, la mAP sí considera los casos donde el modelo no detecta todos los objetos presentes en la imagen.

## 2.2. Despliegue del modelo

Tras platicar de los problemas que estábamos teniendo para hacer inferencia en la RPi a FPS aceptables con nuestro profesor, eventualmente nos dimos cuenta que pudiéramos probar haciendo inferencia cada cierto número de frames, en lugar de hacer inferencia en cada frame, y hubo una mejora en los FPS promedio. Modificamos el algoritmo, y Andrés fue a realizar una prueba para registrar resultados.

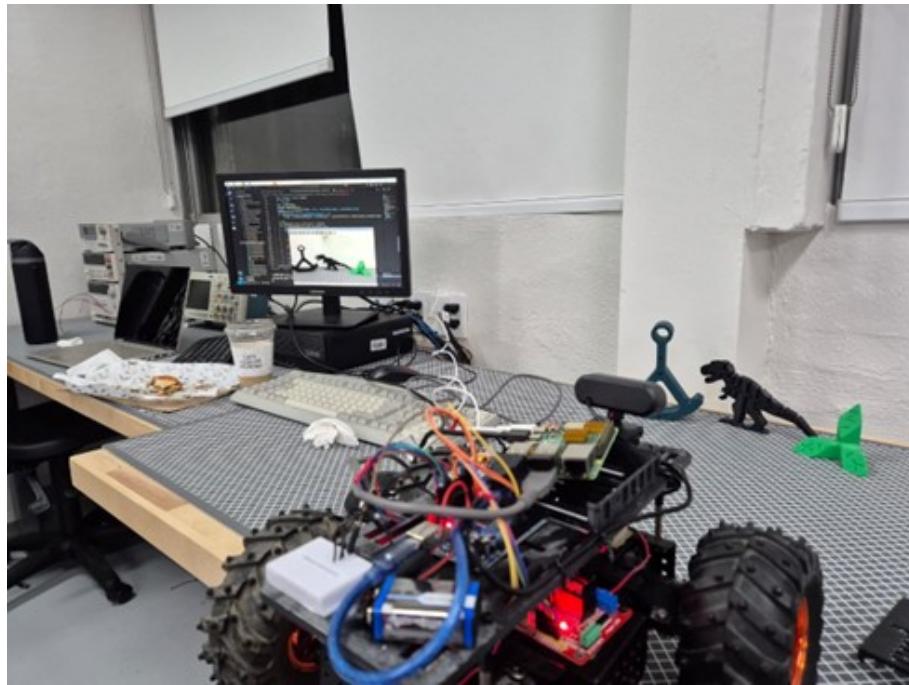


Figura 8: Evidencia de prueba de inferencia en RPi

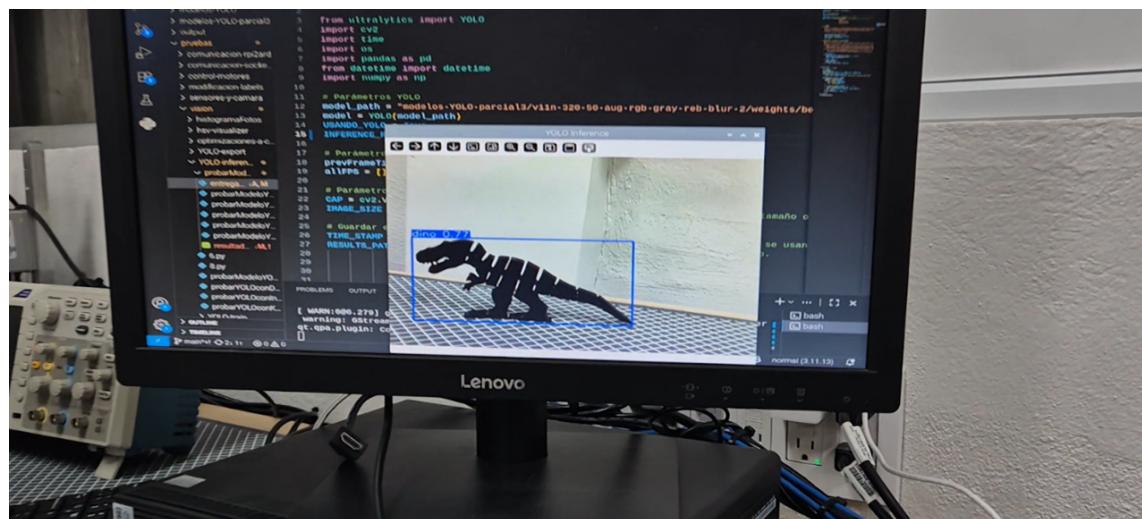


Figura 9: Detección de dinosaurio en RPi

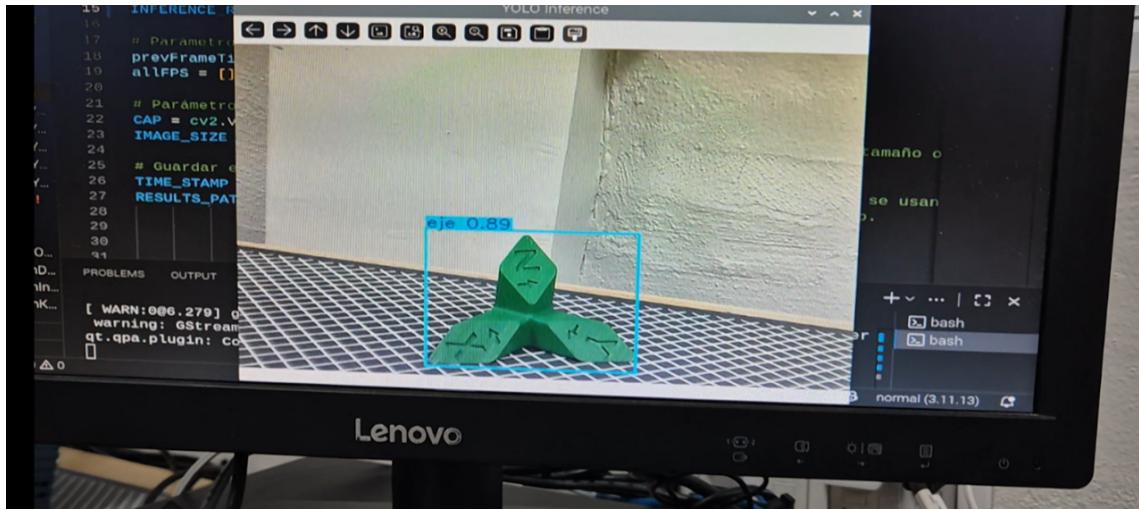


Figura 10: Detección de eje coordenado en RPi

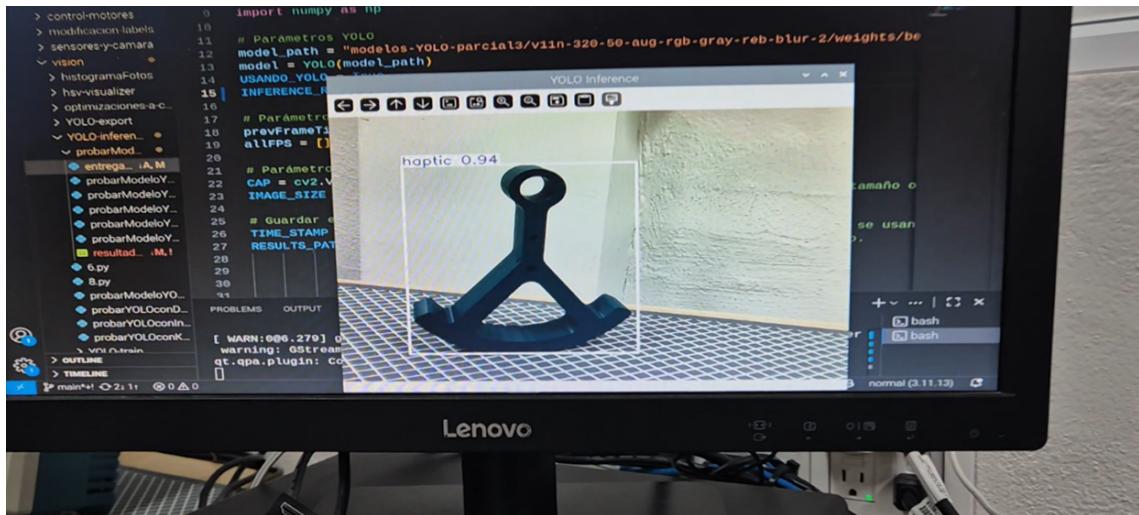


Figura 11: Detección de haptic paddle en RPi

En ella Andrés registró los siguientes resultados:

Inferencia	Nº objetos detectados	Nº frames entre inferencia	FPS promedio*
No	0	N/A	25.01
Sí	3	10	25.06
Sí	3	1	3.02

Tabla 1: Tabla de resultados de pruebas de inferencia en RPi. \*En su momento creo que se estaba calculando la mediana de los FPS en lugar de la media, pero en pruebas posteriores se corroboró que el promedio era similar a la mediana para inferencia cada cierto número de frames.

Estas pruebas se hicieron con el modelo de 320x320 porque queríamos en su momento

priorizar velocidad, pero eventualmente nos dábamos cuenta que no había mucha diferencia en FPS entre los modelos de 320x320 y 640x640 al hacer inferencia cada cierto número de frames.

Estos resultados fueron muy prometedores, porque para nuestra aplicación no era necesario hacer inferencia en cada frame (como en aplicaciones de seguimiento de objetos), entonces fue más que suficiente para cumplir con los requerimientos de nuestro proyecto. A la hora de echar a andar el robot, la detección funcionó como se esperaba.

### 3. Conclusiones

El modelo que entrenamos fue apto para la aplicación para la que fue pensado: la detección de 3 objetos en un cuarto mientras un robot se mueve siguiendo una rutina estilo *roomba* a, como mínimo, 12 FPS. En nuestro caso, aproximadamente 25 FPS.

Fue retadora meterse a todo un ecosistema de entrenamiento, evaluación, y optimización de redes neuronales para detección de objetos, considerando que anteriormente solo había trabajado con clasificación. Implicó adentrarse en documentación de Ultralytics y Roboflow, hacer mucha experimentación y pruebas. Ayudó a lo largo del proceso tener ordenados los componentes del proyecto (scripts de segmentación, comandos para el entrenamiento, scripts para el despliegue en la RPi, carpetas destinadas para cada modelo).

Queda como trabajo futuro:

1. Hacer un mejor benchmarking al entrar en nuevas arquitecturas y aplicaciones. Invertí tiempo entrenando modelos de la versión 8 de YOLO cuando la 11 ya estaba disponible, y también después de tiempo me di cuenta que realmente en la RPi es difícil hacer inferencia a más de 10 de FPS con inferencia continua en la RPi usando Python, y gente con mejores resultados lo ha hecho escribiendo en C++ e incluso *overclockeando* la RPi [3].
2. Agregar imágenes al conjunto de datos que no contengan ninguno de los objetos a detectar, con sus respectivos archivos de anotación vacíos, para reducir falsos positivos.
3. Entrenar YOLO de forma que se pueda usar en producción sin meterse en temas de copyright. Si bien Ultralytics es open source, para uso comercial se requiere comprar una licencia.
4. Volver a probar formatos de exportación de los modelos en dispositivos móviles con mayor capacidad de cómputo.

En general, esta etapa del proyecto fue muy retadora, pero sirvió de mucho aprendizaje para mí en relación con visión computacional y con aprendizaje profundo.

## 4. Referencias

- [1] A. Mendoza, A. Lozano, N. Garza y J. Amaro, «Entrenamiento de pieza con YOLO,» 2025.
- [2] *Is it OK to include images with no labels in the training data set?* <https://github.com/ultralytics/ultralytics/issues/7981>, 2024.
- [3] Politiek, *Deep learning examples on Raspberry 32/64 OS*, <https://qengineering.eu/deep-learning-examples-on-raspberry-32-64-os.html>, 2023.