Proyecto Fin de Máster
Doble Máster de Ingeniería Industrial y Electrónica, Robótica y Automática

Redes Neuronales aplicadas a la detección de posición y direccionamiento de elementos de fabricación

Autor: Francisco Javier Cebolla Verdugo

Tutor: Juan Manuel Escaño González

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

Proyecto Fin de Máster

Doble Máster de Ingeniería Industrial y Electrónica, Robótica y Automática

Redes Neuronales aplicadas a la detección de posición y direccionamiento de elementos de fabricación

Autor:

Francisco Javier Cebolla Verdugo

Tutor:

Juan Manuel Escaño González

Profesor Titular

Dpto. Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2023

Proyecto Fin de Máster: Redes Neuronales aplicadas a la detección de posición y direccionamiento de elementos de fabricación

Autor: Francisco Javier Cebolla Verdugo
Tutor: Juan Manuel Escaño González

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

En agradecimiento total hacia todas las personas que me han apoyado durante los años que he estado realizando mis estudios y a formar a la persona que soy a día de hoy.

F. Javier Cebolla Verdugo

Sevilla, 2023

Resumen

En estos últimos años, ha habido un gran avance respecto a las inteligencias artificales mediante el uso de redes neuronales. El proyecto trata, mediante el uso de esta herramienta aplicada a visión,se analizarán varios tipos de redes neuronales que pueda identificar la posición de una o varias bandejas dentro de una célula de fabricación, y su dirección a través de los carriles que esta tiene, todo ello para transmitirlo a su gemelo digital en el mundo virtual. La identificación de los valores reales de posición y velocidad, dotarán al gemelo digital de la sincronización adecuada para ajustar sus parámetros y tener así una réplica de la realidad, constituyendo así la actualización del modelo.

Abstract

In recent years, there has been significant advancement in artificial intelligence through the use of neural networks. This project, using this tool applied to vision, will analyze various types of neural networks capable of identifying the position of one or more trays within a manufacturing cell, and their direction along the tracks it has, all in order to transmit this information to its digital twin in the virtual world. The identification of the real values of position and speed will provide the digital twin with the appropriate synchronization to adjust its parameters, thereby creating a replica of reality and constituting an update of the model.

Índice

Resumen III
Abstract V

1 Introducción 1
1.1 Motivación del problema 1
1.2 Antecendentes 2
1.3 Gemelos digitales 5
1.4 El problema del aprendizaje 7
1.5 La visión artificial en la industria 8

2 Arquitectura del sistema 11
2.1 Dispositivos Hardware 11
2.1.1 Soporte para Cámara 11
Diseño 11
Impresión 3D 13
Ensamblaje y Resultado Final 14
2.2 Dispositivos Hardware 14
2.2.1 Raspberry Pi 15
2.2.2 PiCamera 15
Portátil de Alta Gama 15
2.2.3 Solid Edge 2020 16
2.2.4 Ultimaker Cura 16
2.2.5 Sistemas Operativos 16
2.2.6 Python 17
2.2.7 Pytorch 17
2.2.8 Pycharm 17
2.2.9 Anaconda 17
2.2.10 OpenCV 17
2.2.11 YOLO (You Only Look Once) 17
2.2.12 LabelIMG 17
2.2.13 CUDA (Compute Unified Device Architecture) 17

3 Metodología y aplicación de algoritmos 19

3.1 Detección mediante machine learning 19
3.1.1 Conceptos generales 19
3.1.2 Datos para entrenamiento 20
3.1.3 SSD (Single Shot MultiBox Detector) 22
3.1.4 YOLO (You Only Look Once) 23
Procedimiento de Entrenamiento con YOLO 25
3.2 Tracking de objetos 26
3.2.1 Conceptos generales 26
3.2.2 SORT 27
3.3 Introducción al Posicionamiento en el Mundo Real 28
3.3.1 Conceptos generales 29
3.3.2 Calibración de la cámara 30
Método de calibración 30
3.3.3 Posicionamiento 3D 30
Implementación de solvePnP 30
3.3.4 Posicionamiento mediante homografía 31
Cálculo de la matriz de homografía 31

4 Resultados 33

5 Conclusiones 37

Apéndice A Códigos en Python 39
A.1 Código para obtener fotogramas para dataset 39
A.2 Código para obtener pasar formato VOC a formato YOLO 41
A.3 Código para entrenamiento mediante SSD 42
A.4 Código para verificación por vídeo mediante SSD 46
A.5 Código para implementación SORT 48
A.6 Clases SORT y KalmanBoxTracker 50
A.7 Detector de tabla ArUco para calibración de la cámara 56
A.8 Código completo para detección,rastreo y posicionamiento de bandejas en video 58

Índice de Figuras 65
Índice de Tablas 67
Bibliografía 69

1 Introducción

Lo que todos tenemos que hacer es asegurarnos de que estamos usando la IA de una manera que sea en beneficio de la humanidad, no en detrimento de la humanidad.

Tim Cook, 2011

En este trabajo, el objetivo será implementar una de las tecnologías que está revolucionando actualmente el campo de la inteligencia artificial (IA), que son las redes neuronales. Estas, gracias a su versatilidad, aprendizaje y complejidad, unido al avance en la computación y el abaratamiento de costes de la industria para realizar más cálculos a mayor velocidad, ha hecho que actualmente todo el mundo esté interesado en ella. Se va a aplicar en este caso, al campo de la visión artificial, para la detección del posicionamiento y dirección de un objeto en un entorno controlado.

1.1 Motivación del problema

En la era de la automatización y la tecnología avanzada, la búsqueda de soluciones que permitan agilizar y mejorar los procesos industriales se ha convertido en una prioridad. La automatización de tareas, la simulación en entornos virtuales y la posterior aplicación en el mundo real ofrecen un sinfín de oportunidades para optimizar la eficiencia y reducir costos. Esta motivación se basa en la creciente necesidad de avanzar hacia una mejora en la producción y en las ventajas que se obtiene al poder realizar pruebas en entornos virtuales frente a entrenamientos en el mundo real.

El desarrollo de gemelos digitales, que simulan entornos del mundo real, y la aplicación de técnicas de aprendizaje automático para entrenar modelos virtuales son dos áreas de interés clave en este contexto. La posibilidad de crear una representación virtual de sistemas físicos complejos y la capacidad de entrenar algoritmos de aprendizaje automático en este entorno ofrece una plataforma ideal para experimentar, probar y perfeccionar soluciones antes de su aplicación en el mundo real.

Este proyecto se inspira en la idea de aprovechar la tecnología actual para realizar pruebas de concepto aplicadas a casos posiblemente reales. La capacidad de simular y entrenar sistemas en un entorno virtual proporciona la flexibilidad necesaria para explorar soluciones de manera segura y eficaz. El objetivo es demostrar cómo la visión artificial y el aprendizaje automático pueden

desempeñar un papel fundamental en la automatización de procesos industriales y en la resolución de problemas complejos.

La motivación principal radica en la posibilidad de diseñar, probar y validar soluciones en un entorno controlado y virtual, antes de llevarlas al mundo real. Esta aproximación reduce riesgos y costos, al tiempo que acelera la implementación de soluciones innovadoras. El resultado de este proyecto es una prueba de concepto que ilustra el potencial de la tecnología de gemelos digitales y su aplicación en situaciones del mundo real.

Este enfoque combina la creatividad y la tecnología para abordar desafíos complejos, al tiempo que sienta las bases para futuras implementaciones exitosas en la industria. El proyecto se centra en el cruce entre la simulación, el aprendizaje automático y la automatización, con el objetivo de inspirar y demostrar las infinitas posibilidades que ofrece la automatización y la tecnología avanzada en el mundo actual.

Esta motivación impulsa la investigación y el desarrollo en la búsqueda de soluciones que mejoren la eficiencia, la precisión y la innovación en la industria.

1.2 Antecendentes

Aunque lo cierto es que el desarrollo avanzado de la inteligencia artificial es algo novedoso, lo cierto es que el desarrollo de máquinas con capacidad para pensar o con capacidades humanas no es relativamente nuevo. Desde hace miles de años el ser humano ha soñado y realizado máquinas que sean capaces de automatizar procesos o tareas para realizar trabajos de manera autónoma, todo ello para ayudar al ser humano en dedicar tiempo en otros motivos o asuntos que sean más productivos que la tarea a automatizar. Tenemos el caso de "La Ilíada", de Homero, en el que se describe a dos sirvientas de oro, al servicio de su dueño y señor, una primera imagen de lo que a día de hoy se conocería como un "androide" o "robot humanoide". Otro ejemplo sería el libro "Automáta", escrito por Herón de Alejandría, en el que describe máquinas capaces de realizar tareas automáticamente, como la apertura y cierre de unas puertas mediante el uso de fuego.

Figura 1.1 Puerta automática de Nerón.

No es hasta llegado el siglo XX, que Alan Turing, el padre de la inteligencia Artificial, sentó las bases de la misma en su artículo "Computing and Intelligence", de la cual salió la famosa prueba de Turing, en la que una persona interacciona mediante texto y debe de discernir si su interlocutor se trata de una persona, o una máquina. Si no es capaz de hacerlo, significa que la máquina ha superado la prueba, y se puede considerar "inteligente".

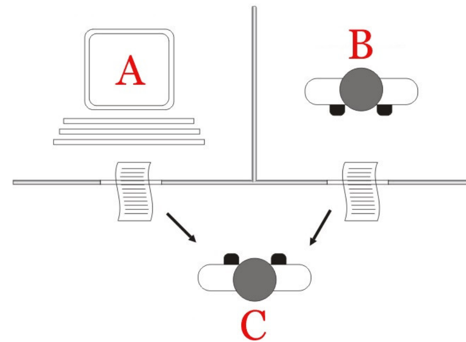

Figura 1.2 Prueba de Turing.

Durante la década de los 50, se considera la época en la que se fundaron las bases de la inteligencia artificial. Durante este período se construyó la primera red neuronal computacional, y, en 1958, Frank Rosenblatt definió las bases del perceptrón. El perceptrón es un elemento que, de unas entradas binarias, produce una sola salida binaria. Para calcular dicha salida, Rosenblatt introduce el concepto de "peso", que es un número real que multiplica a una de las entradas, que expresa la importancia de la respectiva entrada con la salida. Realizando la suma de la multiplicación de pesos por cada entrada, se compara el resultado con un "umbral", si la suma es mayor, el resultado de la salida será 1, y si es menor, 0.

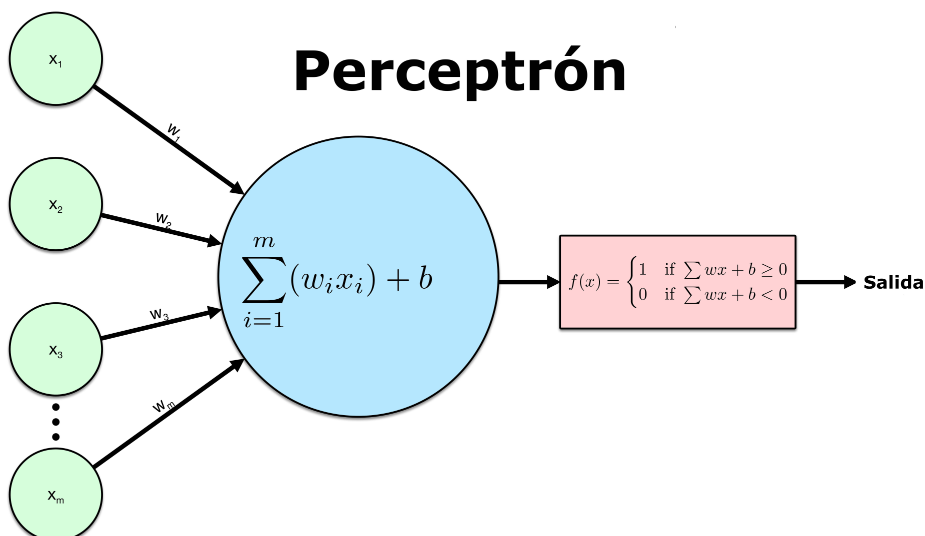

Figura 1.3 Modelo teórico del perceptrón.

Tras unos años, se modificó el perceptrón mejorándolo, introduciéndole el concepto de "multicapa", en el que se añade una capa oculta. Por lo tanto, se trata de una red neuronal de n entradas, m neuronas en su capa oculta, y una neurona de salida. Con este método, se consigue realizar procesos con mucha mayor complejidad, y a su vez, mayor versatilidad. El problema de este tipo de redes neuronales es que cada peso se añadía de manera manual, por lo que, a mayor número de neuronas, mayor complejidad manual.

No fue hasta la década de los años 80 y 90 que se volvió a tener un avance significativo en este ámbito, con el salto de a las neuronas sigmoideas, las cuales en vez de usar una función umbral, usan una función de activación, que en este caso, fuese entre 0 y 1. Existen a día de hoy muchos tipos de funciones de activación, pero esta en concreto, fue la inicial y que se popularizó enormemente.

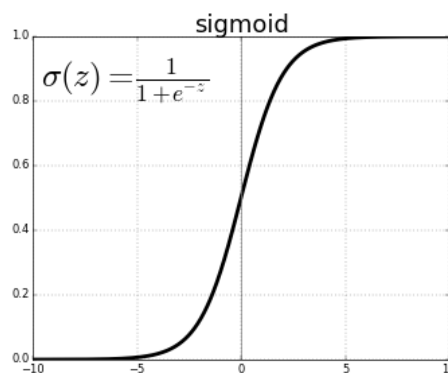

Figura 1.4 Función de activación sigmoide.

El siguiente avance importante en la materia fue las reglas de aprendizaje, o conocido como "backpropagation" en la que la red trabaja bajo aprendizaje supervisado, y necesita un conjunto de entrenamiento que le describa cada entrada y su valor de salida esperado. Con esto, se compara el valor de esperado con el resultado obtenido de un con, por lo que la diferencia de ambas es el error. El objetivo principal de este método es ajustar los pesos para minimizar el error cuadrático medio total después de haber presentado a la red el conjunto de entrenamiento. Se consigue con esto, que la red neuronal "aprenda" a minimizar el error, que una vez entrenada, al recibir un conjunto de datos test, realizará su predicción sobre lo entrenado previamente.

La primera red neuronal convolucional fue desarrollada por Yann LeCun en 1989 y se llamaba LeNet. El trabajo de LeCun sobre las CNNs realmente comenzó a tomar forma con el proyecto LeNet-5, una red neuronal convolucional diseñada para reconocer dígitos escritos a mano. Esto fue desarrollado durante su tiempo en Bell Labs y fue utilizado por el sistema de lectura de cheques de los Estados Unidos en la década de 1990.

LeNet-5 es una red muy simple en comparación con las arquitecturas modernas, pero introdujo varios conceptos importantes que aún se utilizan en las CNNs contemporáneas. Algunas de estas innovaciones incluyen el uso de capas convolucionales (que son excelentes para identificar características locales en las imágenes, como bordes o formas), la submuestreo (reducción de la resolución de una imagen para hacerla más manejable para el procesamiento) y el uso de funciones

de activación no lineales.

Los principales problemas que tuvo esta en aquella época era el escaso poder de computación y almacenamiento de datos que tenían disponibles. En la siguiente figura se muestra el número de transistores integrados en un microprocesador, a escala logarítmica, y vemos que prácticamente se cumple la ley de Moore, aunque a día de hoy se ha reformulado ya que la compresión de transistores en más complicada por el tamaño alcanzado, el orden de magnitud desde la década de los 90 hasta los tiempos recientes se encuentra entorno a cien mil o un millón de veces los transistores que alberga un microprocesador actual respecto a un microprocesador del año 1990.

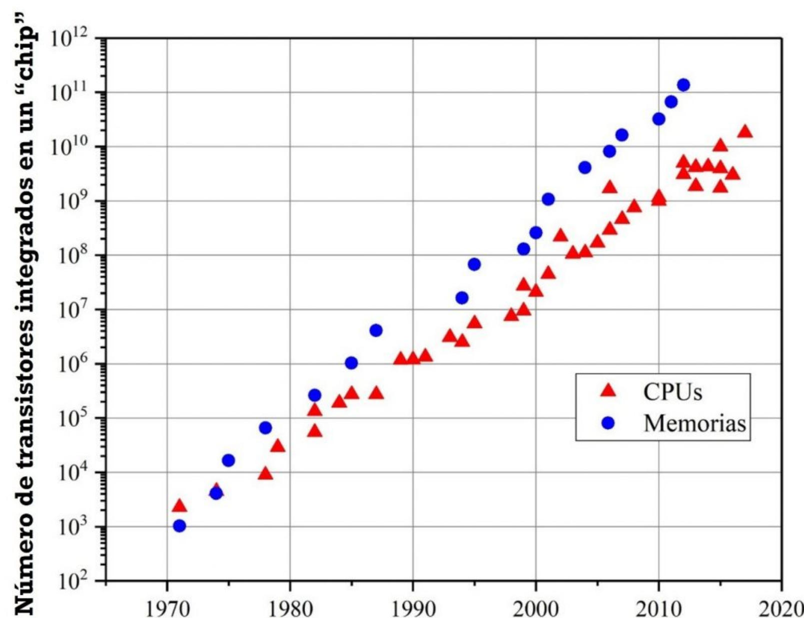

Figura 1.5 Ley de Moore para CPU y memorias.

Respecto al almacenamiento y las memorias, ocurre prácticamente la misma diferencia, mientras que en aquel entonces el almacenamiento se encontraba entorno a los bytes o kbytes, a día de hoy, cualquier ordenador puede alberga 1 o 2 terabytes sin problemas, alrededor de 1 billón (millón de millones) o mil billones más de información puede tener un ordenador respecto a uno de hace 30 años.

1.3 Gemelos digitales

Junto a la revolución digital, unos de los campos que también ha avanzado enormemente ha sido el concepto de "gemelos digitales". Este concepto se refiere a la creación de réplicas digitales precisas de entidades físicas, que pueden abarcar desde objetos individuales hasta sistemas completos y entornos complejos. Estos modelos digitales dinámicos, que son actualizados en tiempo real, permiten simular, predecir y visualizar cambios en sus contrapartes físicas.

El concepto de gemelo digital se refiere a la creación de un modelo digital de un elemento de fabricación observable (OME, por sus siglas en inglés). Esta representación digital es capaz de

reflejar en tiempo real el estado y las actividades del elemento físico correspondiente, permitiendo una interacción y sincronización continua entre el mundo físico y el digital. Se originó en la industria aeroespacial de la NASA, y ha ganado impulso en los últimos años debido a la convergencia de tecnologías avanzadas como Internet de las Cosas (IoT), Big Data, Inteligencia Artificial (IA), Machine Learning y la computación en la nube. Este trabajo pretende explorar su aplicabilidad y las posibles ventajas que nos pueden otorgar la aplicación de gemelos digitales, así como los desafíos técnicos que con el se encuentran.

Los gemelos digitales se han utilizado durante décadas en sectores de alta exigencia, como la aeroespacial, para simular el rendimiento de las naves espaciales en entornos adversos. Sin embargo, es solo recientemente, gracias a la expansión de la capacidad de procesamiento de datos y el desarrollo de algoritmos de aprendizaje automático más sofisticados, que su uso se ha extendido a otros sectores. Las aplicaciones de los gemelos digitales en la fabricación son diversas, incluyendo, pero no limitándose a, el control en tiempo real, el análisis fuera de línea, el mantenimiento predictivo, la verificación de la salud de los equipos y el diseño de ingeniería asistido por computadora. Estas aplicaciones no solo mejoran la eficiencia de los procesos de fabricación, sino que también permiten una mayor flexibilidad y adaptabilidad ante los cambios en las demandas del mercado y las condiciones operativas.

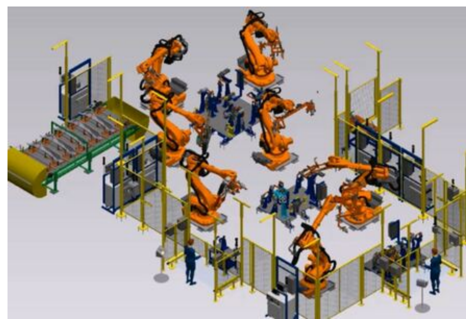

Figura 1.6 Gemelo digital de un proceso de fabricación.

La serie de normas ISO 23247:2021 [1, 2, 3, 4] establece los principios generales para el desarrollo de gemelos digitales en la fabricación. Estos principios incluyen la precisión en la representación de datos, una comunicación efectiva entre el modelo digital y su contraparte física, y la integridad y seguridad de los datos. También se enfatiza la importancia de la extensibilidad y la granularidad de los modelos digitales, permitiendo su adaptación y escalabilidad en diversos entornos de fabricación.

La implementación de un gemelo digital requiere un enfoque sistemático y considerado, con un claro entendimiento de los objetivos y el alcance del proyecto. Los desafíos en la implementación de gemelos digitales incluyen la integración de sistemas heterogéneos, la gestión de grandes volúmenes de datos y la necesidad de mantener una sincronización precisa entre los modelos digitales y sus contrapartes físicas.

La norma ISO 23247-4 [4] proporciona ejemplos específicos y casos de uso de gemelos digitales en la fabricación. Uno de estos ejemplos es la optimización de las operaciones de remoción de material, donde los gemelos digitales del producto, el proceso y el equipo enriquecen el flujo de datos desde

la ejecución del proceso con información contextual sobre el plan de proceso, las herramientas de corte utilizadas y la parte que se está mecanizando y midiendo. Este caso de uso ilustra cómo los gemelos digitales pueden facilitar un bucle de retroalimentación entre el proceso ejecutado y el planificado, mejorando la eficiencia y la calidad del proceso de fabricación.

Además de los usos y principios generales ya mencionados, la norma ISO 23247-1:2021 [1] profundiza en el marco de los gemelos digitales en la fabricación, abordando varios aspectos clave. Uno de ellos es la necesidad de una sincronización precisa entre el gemelo digital y el OME, lo que asegura que los sistemas de fabricación estén constantemente optimizados. Los gemelos digitales reciben información de rendimiento en tiempo real del sistema físico, lo que permite una mejora continua y adaptación del proceso de fabricación.

En este contexto, los gemelos digitales de engranajes y correas en una célula de fabricación integran datos de sensores y sistemas de monitoreo para proporcionar una visión precisa del estado actual y la tasa de desgaste de estos componentes. Estos modelos digitales dinámicos permiten predecir la vida útil restante de los componentes y programar el mantenimiento preventivo de manera más efectiva, minimizando así las interrupciones no planificadas en la producción.

La implementación de este tipo de gemelos digitales requiere una sincronización precisa entre el modelo digital y los componentes físicos. Los datos recopilados a través de sensores en la célula de fabricación se utilizan para alimentar y actualizar constantemente el gemelo digital, asegurando que la representación virtual refleje con exactitud el estado actual de los engranajes y correas.

Este enfoque no solo mejora la fiabilidad y la eficiencia de las células de fabricación, sino que también proporciona datos valiosos para la optimización continua del diseño y la operación de estos componentes. Con la ayuda de análisis avanzados y simulaciones, los gemelos digitales pueden identificar patrones de desgaste y predecir posibles fallos, permitiendo a los operarios tomar decisiones informadas sobre el mantenimiento y la operación de la maquinaria.

La norma ISO 23247-1:2021 [1] subraya la importancia de la precisión, la sincronización y la flexibilidad en la creación de gemelos digitales, así como la necesidad de adaptarse a diferentes aplicaciones y contextos dentro del ciclo de vida del producto. Esta implementación específica de gemelos digitales para monitorear el desgaste en engranajes y correas demuestra cómo la tecnología puede ser aplicada para mejorar la eficiencia y la fiabilidad en los procesos de fabricación.

1.4 El problema del aprendizaje

El aprendizaje es un proceso intrínseco a la inteligencia, tanto natural como artificial. En el contexto de la inteligencia artificial, el aprendizaje se refiere a la capacidad de un sistema para mejorar su desempeño en una tarea específica a través de la experiencia. Este proceso es central en áreas como el aprendizaje automático (machine learning) y el aprendizaje profundo (deep learning), donde los algoritmos ajustan sus parámetros internos, generalmente pesos en una red neuronal, para minimizar una función de coste o maximizar una función de recompensa.

En el aprendizaje supervisado, se proporciona al sistema un conjunto de ejemplos etiquetados, y el objetivo es aprender una función general que mapee las entradas a las salidas deseadas. Sin embargo, el aprendizaje no supervisado intenta encontrar patrones y estructuras ocultas en datos no

etiquetados. Por otro lado, el aprendizaje por refuerzo se centra en cómo los agentes deben tomar acciones en un entorno para maximizar alguna noción de recompensa acumulativa.

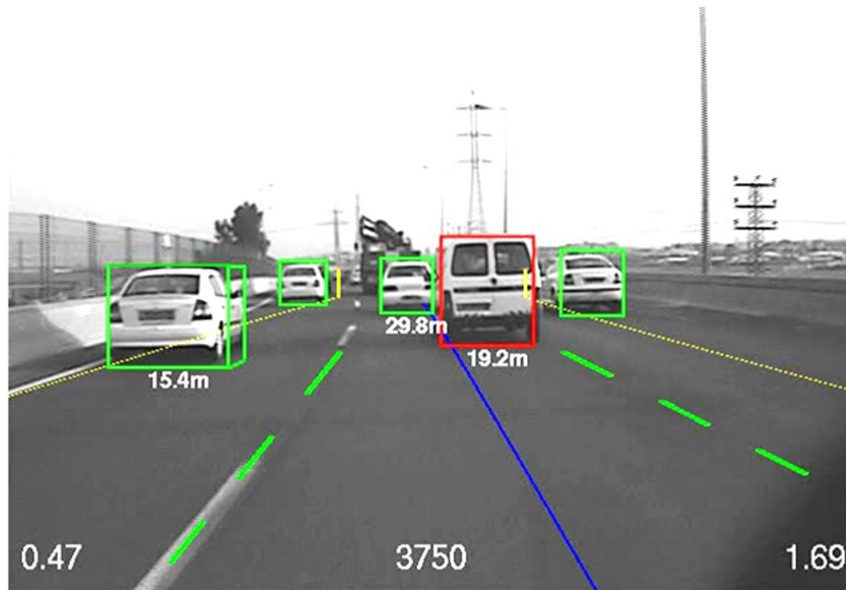

Figura 1.7 Detección de vehículos en la carretera y su posición respecto a la cámara.

El "problema del aprendizaje" abarca varios desafíos, como el sobreajuste, donde un modelo se ajusta demasiado a los datos de entrenamiento y no generaliza bien a datos nuevos; la selección de características, que implica determinar qué información es relevante para la tarea; y la escalabilidad, ya que los algoritmos deben ser capaces de aprender eficientemente a medida que la cantidad de datos crece. Además, está el desafío de la transferencia de aprendizaje, donde un sistema debe aplicar lo aprendido en un contexto a tareas relacionadas pero diferentes.

1.5 La visión artificial en la industria

La visión artificial, un campo interdisciplinario que involucra el procesamiento y análisis de imágenes, es fundamental en la automatización industrial. En la fábrica moderna, los sistemas de visión artificial son empleados para una variedad de aplicaciones que van desde la inspección de calidad y la robótica hasta la guía de máquinas y la identificación de partes.

En inspección de calidad, los sistemas de visión artificial proporcionan capacidades de inspección no destructiva en tiempo real, lo que permite detectar defectos que no serían perceptibles para el ojo humano. Esto no solo mejora la calidad del producto sino que también reduce los costos al minimizar el desperdicio y las devoluciones.

La integración de la visión artificial en sistemas robóticos, conocida como robótica guiada por visión, permite que los robots realicen tareas complejas como el ensamblaje de precisión, la soldadura y la pintura con gran eficiencia y repetibilidad. Además, la visión artificial es crucial en la logística para la clasificación y el rastreo de productos, facilitando sistemas de manejo de materiales automatizados y eficientes.

El avance en algoritmos de aprendizaje profundo ha ampliado aún más las capacidades de la visión artificial en la industria, permitiendo la identificación y clasificación avanzadas, así como la habilidad de adaptarse a variaciones en los procesos de producción. Sin embargo, la implementación de estos sistemas enfrenta desafíos tales como la necesidad de grandes conjuntos de datos etiquetados, la computación de alta potencia y la integración con sistemas de control existentes.

Figura 1.8 Imagen de una manzana con un filtro para detectar irregularidades.

2 Arquitectura del sistema

En la era actual, donde la tecnología avanza a pasos agigantados, se ha vuelto cada vez más esencial contar con proyectos innovadores que aprovechen al máximo los recursos disponibles. En este contexto, la creación de proyectos que integren elementos físicos y tecnológicos se ha vuelto una tarea crucial para el desarrollo y la evolución de diferentes industrias y áreas de estudio. En este apartado, se realizará descripción de los elementos fundamentales utilizados en la realización de este trabajo, tanto desde la perspectiva de las necesidades físicas para la colocación de la visión sobre el espacio de trabajo como de los dispositivos hardware y software necesarios durante todo el proceso.

En este trabajo, es fundamental comenzar por identificar las necesidades físicas respecto a la visión del espacio en el que se llevará a cabo el proyecto. La adecuada disposición de la cámara en el lugar de trabajo es un aspecto primordial que hay que tener en cuenta para no cometer errores que no sean subsanables mediante el procesamiento de los datos obtenidos. Una correcta distribución del espacio, iluminación, sombras y/o defectos en la infraestructura son elementos clave para alcanzar los objetivos propuestos.

2.1 Dispositivos Hardware

2.1.1 Soporte para Cámara

Diseño

Para el trabajo, se ha diseñado ajustable y versátil para la cámara de la Raspberry Pi 4. Su función es mantener la cámara fija y estable, permitiendo obtener una visión completa de la célula de fabricación. Gracias a su diseño, se debe de garantizar una referencia de posición constante para la cámara, lo que facilita futuras captura de datos y asegura la compatibilidad de datos. El soporte es adaptable en altura y ángulo de visión, lo que proporciona flexibilidad para configurarlo en caso de necesidad de cambio de punto de visión. Con la inclusión de tres pasadores roscados como ejes, es posible realizar ajustes precisos en la posición final de la cámara, optimizando la obtención de resultados confiables. Además, el soporte alberga debajo del mismo la Raspberry Pi 4, brindando una solución organizada y completa para la captura de datos y análisis detallado de los procesos de fabricación. En conjunto, el diseño del soporte y su ubicación estratégica de la cámara garantizan un sistema eficiente y funcional para la toma de datos y de imágenes de la célula.

Para el diseño del prototipo, se ha usado el programa de diseño asistido por ordenador "Solid Edge", versión 2020. En dicho programa, se va a realizar la configuración de un dispositivo de 4 piezas para lograr el objetivo de tener un soporte eficaz y útil, imprimible físicamente en una impresora 3d, que cumpla con su objetivo principal: Obtención de imágenes óptimas para la detección de las bandejas. Para esto, se realizará el conjunto de este prototipo de 4 piezas.

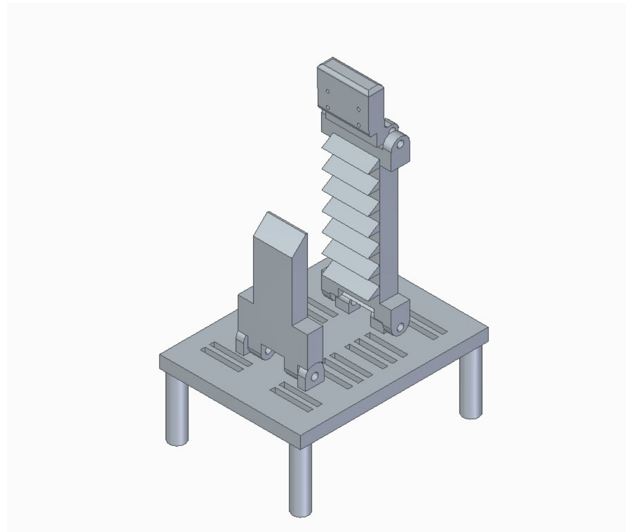

Figura 2.1 Conjunto de piezas.

- **Cámara-Efector**: Es el punto en que se va a anclar la cámara, y se va a regular el ángulo de la cámara para poder ver con mejor precisión la célula.

- **Brazo**: Es el brazo que extiende a la cámara a un punto óptimo de visión respecto a la Raspberry Pi 4, necesitamos que tenga unos apoyos sobre el propio recorrido de la pieza que encaje con el apoyo, para que no se caiga la pieza. Esta a su vez tiene en sus extremos 2 pasadores para que pase una varilla roscada, la cual hará de eje para poder regular el ángulo de la cámara.

- **Apoyo**: Esta pieza cumple la función de actuar como contrapeso del brazo en el diseño del soporte para la cámara. Esta pieza se coloca sobre la base y presenta un extremo achaflanado que permite un encaje preciso con los apoyos creados en el brazo. Al proporcionar estabilidad y equilibrio, el apoyo asegura que el brazo se mantenga firme durante el funcionamiento del sistema, garantizando así una posición constante y precisa para la cámara. Su diseño estratégico contribuye significativamente a mantener una visión óptima de la célula de fabricación y a obtener datos fiables durante el experimento.

- **Base**: Es la "mesa" que tiene una altura libre suficiente sobre el suelo para que la Raspberry Pi 4 se encuentre alojada en su interior, y no esté muy pegada a la misma para no generar más calor. Tiene 4 pasadores para que se coloque un extremo del apoyo y el extremo del brazo, para que la distancia en la base sea fija, y la altura se regule en función de donde queda colocado el punto de conexión entre el apoyo y el brazo.

La varilla roscada usada en este caso es de 3mm, y se ha tomado como restricción a tener en cuenta la longitud del cable de la PiCamera, ya que en este caso son 140mm, teniendo en cuenta que la conexión del efector final del soporte y la distancia del conector de la Raspberry Pi 4 no debe de ser superior a esta misma.

Impresión 3D

La impresión 3D es una técnica de fabricación aditiva esencial para el desarrollo de prototipos y piezas finales. Utilizando una impresora Ender 3 Pro con material PLA, se han ajustado meticulosamente los parámetros para obtener resultados óptimos:

- **Altura de capa:** Configurada a 0.2 mm para equilibrar la resolución y el tiempo de impresión, proporcionando una superficie lisa y un detalle adecuado en la pieza final.

- **Grosor de pared:** Con 4 capas, se asegura la fortaleza y estabilidad estructural del objeto, vital para su funcionalidad y longevidad.

- **Temperatura de impresión:** Ajustada a 215ºC para el PLA, esta temperatura garantiza una fusión y adhesión intercapas apropiadas, evitando defectos como deformaciones o hilos residuales.

- **Velocidad de impresión:** Establecida en 60 mm/s para mantener una impresión detallada sin comprometer la calidad de la superficie.

- **Temperatura de la placa de impresión:** Mantenida en 60ºC para prevenir el desprendimiento de las capas iniciales y asegurar la adherencia del material durante todo el proceso de impresión.

- **Densidad de relleno:** Seleccionada al 25 %, lo que proporciona un compromiso entre la resistencia estructural y el uso eficiente del material, sin sacrificar la integridad del objeto.

- **Patrón de relleno:** Se ha optado por un patrón de rejilla, que ofrece una buena estabilidad dimensional y es rápido de imprimir, mientras soporta adecuadamente las capas superiores.

El PLA es preferido por su facilidad de uso, baja contracción y excelente adhesión intercapas, haciéndolo ideal para una gran variedad de aplicaciones de impresión 3D. La Ender 3 Pro se ha escogido específicamente por su confiabilidad y consistencia en la calidad de impresión, lo que la convierte en una opción popular entre los aficionados y profesionales de la impresión 3D. La adecuada configuración de estos parámetros ha sido clave para alcanzar la calidad deseada para las piezas del proyecto.

Se muestra en la figura 2.2 la previsualización con el tiempo y las capas ordenadas para poder generar el archivo necesario para imprimir los objetos en la impresora. En esta previsualización se pueden ver el interior de los objetos y los soportes creados, útil para mejorar en algunos casos la forma de imprimir o la metodología para ello:

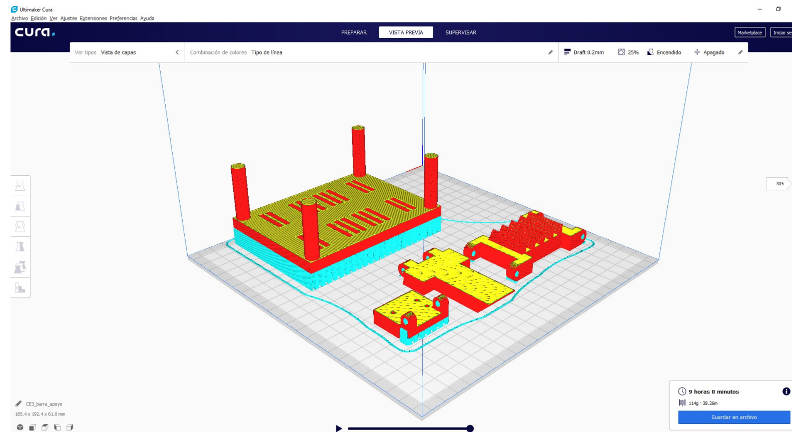

Figura 2.2 Previsualización de los objetos a imprimir.

Ensamblaje y Resultado Final

El ensamblaje de la Raspberry Pi junto con su soporte se muestra en la figura 2.3, creando un conjunto versátil y funcional, que se puede ubicar en cualquier sitio en el que pudiera ubicarse la Raspberry pi 4

Figura 2.3 Imagen del ensamblaje de la Raspberry Pi con su soporte..

2.2 Dispositivos Hardware

En este capítulo, se describen con detalle los dispositivos físicos que constituyen la infraestructura esencial de nuestro proyecto. La selección del hardware adecuado es crítica, ya que cada componente debe cumplir con los requisitos específicos de rendimiento y fiabilidad para garantizar el éxito de las aplicaciones de visión por computadora que se implementarán. Se realizará un análisis de cada dispositivo, incluyendo sus características, especificaciones y el papel que desempeñan en el conjunto del sistema. Este capítulo servirá como referencia para comprender la base sobre la cual

se construye y opera la solución tecnológica propuesta.

2.2.1 Raspberry Pi

La Raspberry Pi es un microordenador de tamaño reducido con una amplia variedad de usos. A continuación, se presentan algunas de sus características clave:

Procesador: La Raspberry Pi está equipada con un procesador ARM que puede funcionar a velocidades de hasta 1500 MHz.

Memoria RAM: El modelo usado tiene 8 GB de RAM.

Conectividad: La Raspberry Pi ofrece múltiples opciones de conectividad, como puertos USB, HDMI dual, Ethernet y en algunos modelos, conexión Wi-Fi y Bluetooth.

Almacenamiento: La Raspberry Pi utiliza tarjetas microSD para el almacenamiento del sistema operativo y datos, en nuestro caso, se ha usado una tarjeta SD de 128 GB.

Limitaciones: Aunque versátil, la Raspberry Pi tiene limitaciones en cuanto a rendimiento, adecuación para juegos y ejecución de aplicaciones de alto rendimiento.En cuanto a GPU, no tiene incorporada, por lo que para aplicaciones sobre imágenes y datos gráficos se usara la CPU.

Sistema Operativo: Se ha usado en el proyecto el sistema operativo Raspbian.

2.2.2 PiCamera

La Raspberry Pi Camera Module 2 es una cámara compacta diseñada para su uso con la Raspberry Pi. Tiene un precio asequible y un tamaño de alrededor de 25 x 24 x 9 mm, con un peso de solo 3 gramos. Ofrece una resolución de imagen de 5 megapíxeles y es capaz de grabar videos en modos de 1080p30, 720p60 y 640 x 480p60/90. Utiliza el sensor OmniVision OV5647 y tiene un área de imagen de 3.76 x 2.74 mm. La cámara Module 2 tiene un enfoque fijo con una profundidad de campo de aproximadamente 1 metro hasta el infinito y una longitud focal de 3.60 mm +/- 0.01.

Portátil de Alta Gama

El portátil utilizado en este proyecto es un dispositivo de alta gama diseñado para manejar tareas de computación intensiva y aplicaciones gráficas avanzadas. A continuación, se detallan sus características más destacadas:

Procesador: Equipado con un Intel® Core™ i7-8750H, el portátil ofrece una frecuencia base de 2.2 GHz y puede alcanzar velocidades significativamente mayores mediante Turbo Boost, apoyado por una caché inteligente de 9MB que mejora la eficiencia del procesamiento.

Memoria RAM: Cuenta con 16 GB de memoria DDR4 a 2133MHz, proporcionando una amplia capacidad para multitareas y operaciones exigentes en memoria.

Almacenamiento: Para el almacenamiento de datos, el portátil combina un disco duro de 1TB que opera a 5400 RPM con un disco SSD NVMe de 256GB, lo que permite un acceso rápido al sistema

operativo y aplicaciones, así como un espacio considerable para almacenamiento de archivos.

Gráficos: El portátil incluye una tarjeta gráfica Nvidia Geforce GTX1070 con 8GB de memoria GDDR5.

Sistema Operativo y Software: El sistema operativo usado es Windows 10.

Este equipo proporciona una potencia y una flexibilidad excepcionales para tareas que van desde el desarrollo de software hasta el procesamiento de imágenes y vídeos, lo que lo convierte en la mejor opción tanto para realizar el entrenamiento de la red neuronal para la detección de bandejas en imágenes como para realizar la mayoría de cálculos que requieran un costo de tiempo inasumible para la rapsberry pi 4.

Se muestra en la siguiente tabla 2.1 una comparativa entre ambos dispositivos para comprensión de la diferencia entre uno y el otro.

Tabla 2.1 Comparación entre el Portátil de Alta Gama y la Raspberry Pi 4.

Característica	**Portátil de Alta Gama**	**Raspberry Pi 4**
Procesador	Intel® Core™ i7-8750H (2.2 GHz)	ARM Cortex-A72 (1.5 GHz)
Memoria RAM	16 GB DDR4	8 GB LPDDR4
Almacenamiento	1TB HDD + 256GB SSD	128 GB microSD
Gráficos	Nvidia Geforce GTX1070 (8GB GDDR5)	Broadcom VideoCore VI
Sistema Operativo	Windows 10	Raspbian
Conectividad	WiFi 802.11 ac, Bluetooth 4.2, Ethernet	WiFi 802.11ac, Bluetooth 5.0, Ethernet
Puertos	HDMI, USB 3.0/3.1 Type-C	HDMI dual, USB 2.0/3.0

Nota: El portátil de Alta Gama soporta CUDA por tener una tarjeta gráfica Nvidia, lo cual es útil para aplicaciones de computación paralela y procesamiento intensivo de datos. La Raspberry Pi 4 no soporta CUDA ya que cuenta con una GPU Broadcom VideoCore VI.

2.2.3 Solid Edge 2020

Solid Edge 2020 es una herramienta de modelado 3D CAD, desarrollada por Siemens. Ofrece diseño mecánico avanzado, simulación, fabricación, y gestión de datos de productos y componentes. Es ampliamente utilizado para la creación de piezas complejas y el ensamblaje de sistemas.

2.2.4 Ultimaker Cura

Ultimaker Cura es un software de impresión 3D de código abierto, considerado uno de los más usados para el corte de modelos 3D. Convierte los modelos en instrucciones de impresión (G-code) y ofrece personalización con ajustes predefinidos o avanzados para controlar cada aspecto de la impresión.

2.2.5 Sistemas Operativos

Se ha usado para este trabajo por un lado el sistema operativo Raspbian para la Rapsberry Pi 4, como Windows 10 para el portátil de alta gama.

2.2.6 Python

Python es un lenguaje de programación de alto nivel, interpretado y con una sintaxis muy legible. Es muy popular en la ciencia de datos, desarrollo web y automatización debido a su facilidad de aprendizaje y la amplia gama de bibliotecas disponibles.

2.2.7 Pytorch

Pytorch es una biblioteca de aprendizaje automático de código abierto para Python, conocida por su flexibilidad y velocidad. Es especialmente popular en investigación y desarrollo de aplicaciones de inteligencia artificial, debido a su capacidad para realizar cálculos complejos de forma eficiente.

2.2.8 Pycharm

PyCharm es un entorno de desarrollo integrado (IDE) utilizado para programar en Python. Ofrece herramientas de codificación inteligente, depuración, pruebas y soporte para desarrollo web con Django.

2.2.9 Anaconda

Anaconda es una distribución de Python y R que simplifica la gestión de paquetes y despliegue. Es ampliamente utilizado en la ciencia de datos y aprendizaje automático para gestionar bibliotecas, dependencias y entornos de trabajo de forma aislada.

2.2.10 OpenCV

OpenCV (Open Source Computer Vision Library) es una biblioteca de programación de código abierto para visión artificial y aprendizaje automático. Proporciona herramientas para procesar imágenes y realizar tareas como detección de objetos, reconocimiento facial y seguimiento de objetos.

2.2.11 YOLO (You Only Look Once)

YOLO es un sistema de detección de objetos en tiempo real que puede identificar objetos en imágenes y vídeos rápidamente. Es conocido por su velocidad y precisión, y es ampliamente utilizado en aplicaciones de visión artificial.

2.2.12 LabelIMG

LabelIMG es una herramienta gráfica de anotación de imágenes. Se utiliza para marcar y etiquetar objetos en imágenes para crear conjuntos de datos que se utilizan para entrenar modelos de detección de objetos.

2.2.13 CUDA (Compute Unified Device Architecture)

CUDA es una plataforma de computación paralela y un modelo de programación inventado por NVIDIA. Permite aumentar el rendimiento de la computación aprovechando la potencia de las unidades de procesamiento gráfico (GPU) para tareas de cálculo intensivo.

3 Metodología y aplicación de algoritmos

Una de las virtudes del ingeniero es la eficiencia.

GUANG TSE

La capacidad para detectar, seguir y posicionar objetos en el mundo real mediante algoritmos de aprendizaje automático y visión por computadora es un desafío significativo en la ingeniería moderna. Este capítulo desglosa el problema en sus componentes fundamentales y describe la metodología y las aplicaciones de algoritmos que permiten a las máquinas interpretar y comprender el mundo visual con precisión y eficiencia.

3.1 Detección mediante machine learning

En esta sección, se presenta de manera concisa la metodología propuesta para la detección de objetos utilizando técnicas de machine learning.

3.1.1 Conceptos generales

Para el entrenamiento efectivo de una red neuronal, se requieren diversos componentes y consideraciones metodológicas. Fundamentalmente, es imprescindible disponer de un conjunto de datos robusto y representativo, compuesto por ejemplos anotados que sirven para que la red aprenda a realizar las tareas deseadas. Existen múltiples métodos de entrenamiento, como la propagación hacia atrás, el algoritmo de optimización de Adam y la regularización, entre otros.

Los *batches* representan subconjuntos del conjunto de datos de entrenamiento que se usan para actualizar los parámetros del modelo en cada iteración. Por su parte, los *epochs* hacen referencia a la cantidad de veces que el algoritmo de aprendizaje trabaja a través del conjunto de datos completo. Un *epoch* incluye tantas iteraciones como sea necesario para que todos los batches sean procesados una vez.

Para la detección de objetos en imágenes, se utilizan técnicas que permiten localizar y clasificar elementos dentro de una imagen digital. Estas técnicas pueden incluir métodos como las redes convolucionales, detectores de bordes, y sistemas de segmentación semántica. Adicionalmente,

se aplican métricas de evaluación como la precisión, el recall y el valor F1 para cuantificar el rendimiento de la red.

Es también crucial abordar el sobreajuste, un fenómeno por el cual la red neuronal se especializa excesivamente en los datos de entrenamiento y pierde capacidad de generalización. Para mitigar este problema, se emplean técnicas como la validación cruzada y el aumento de datos (*data augmentation*).

3.1.2 Datos para entrenamiento

La adquisición de imágenes para entrenar la red neuronal se ha llevado de la siguiente manera: Se ha construido un soporte de madera de 2.6 metros de altura, véase en la figura 3.1, equipado con un enchufe en la parte superior, lo que permitió conectar la Raspberry Pi de forma continua y garantizar un suministro eléctrico ininterrumpido.

Este arreglo facilitó la ubicación estratégica de la Raspberry Pi y la PiCamera para capturar imágenes desde una perspectiva elevada, optimizando así el ángulo de visión y la cobertura del área de interés. Finalmente tras la observación del entorno, se ha optado por colocar la cámara en una posición de 3 metros de altura, para tener una mejor imagen que desde la altura del soporte fabricado, véase la figura 3.2 la posición final de la cámara, y en la figura 3.3 se ve, realizada una fotografía con un teléfono de última generación para obtener una vista general del espacio de trabajo en cuestión.

Figura 3.1 Soporte de madera elevado para colocación de la Raspberry Pi.

Para recopilar un conjunto de datos variado y amplio, se utilizó una alargadera que permitió la movilidad del sistema de captura, asegurando la recopilación de imágenes desde múltiples ubicaciones y bajo distintas condiciones de iluminación. Se realizaron capturas de vídeo utilizando la Raspberry Pi y la PiCamera, las cuales luego se procesaron para extraer fotogramas individuales que sirvieron como ejemplos para el entrenamiento. Este método no solo proporcionó una cantidad significativa de datos sino que también aseguró la diversidad en las muestras, un aspecto crítico para el desarrollo de un modelo de machine learning robusto y generalizable.

Figura 3.2 Posición final de la Raspberry Pi 4 con la cámara.

En concreto, se han realizado entorno a 18 vídeos para poder entrenar a la red neuronal, con una tasa de imagenes de unos 5 fps aproximadamente de una duración media de unos 30-60 segundos dependiendo del experimento en cuestión, se muestra en las figuras 3.4 y figura3.5.

Figura 3.3 Vista general desde altura final con móvil de última generación.

Con esto, se obtiene una base de imágenes de aproximadamente 6.180 fotografías. Se muestra el código para obtener las imágenes con la Raspberry Pi 4 en el apéndice A.1. Para anotar estas imágenes y prepararlas para el proceso de entrenamiento de la red neuronal, se utiliza la herramienta *labelImg*. Esta herramienta es una aplicación gráfica que facilita la tarea de etiquetar objetos en imágenes. El proceso de etiquetado implica cargar cada imagen en *labelImg*‘, y manualmente, se dibujan cuadros delimitadores alrededor de cada objeto de interés en la imagen. Para cada cuadro delimitador, se asigna una etiqueta de clase correspondiente al objeto que se está marcando. Una vez que todas las imágenes están etiquetadas, *labelImg* guarda las coordenadas de los cuadros delimitadores y las etiquetas de clase en archivos XML siguiendo el estándar VOC. Este formato es ampliamente utilizado para el entrenamiento de algoritmos de detección de objetos y resulta ser compatible con muchas bibliotecas de aprendizaje profundo, luego una vez tengamos el dataset completo de las imágenes etiquetado en formato VOC, se realizará una transformación del dataset completo a formato YOLO para poder entrenar la ned neuronal específica en YOLO. En el trabajo se va a realizar la detección por imágenes de las bandejas, que tendrán de nombre clasificador "bandeja", los botones blancos al final de los carriles, se clasificarán como "boton_rojo_blanco", y el botón amarillo que se encuentra en el almacén de las bandejas se clasificará como "boton_rojo_amarillo", como se muestra en la figura 3.5.

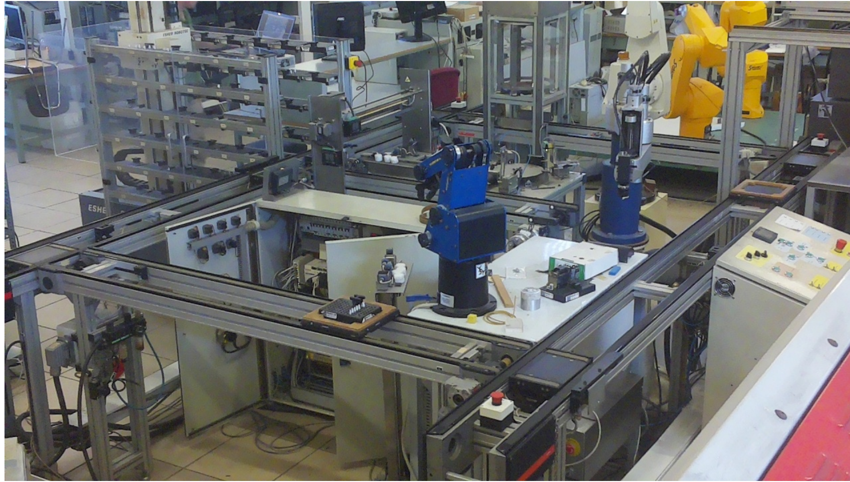

Figura 3.4 Fotograma usado para entrenamiento de red neuronal.

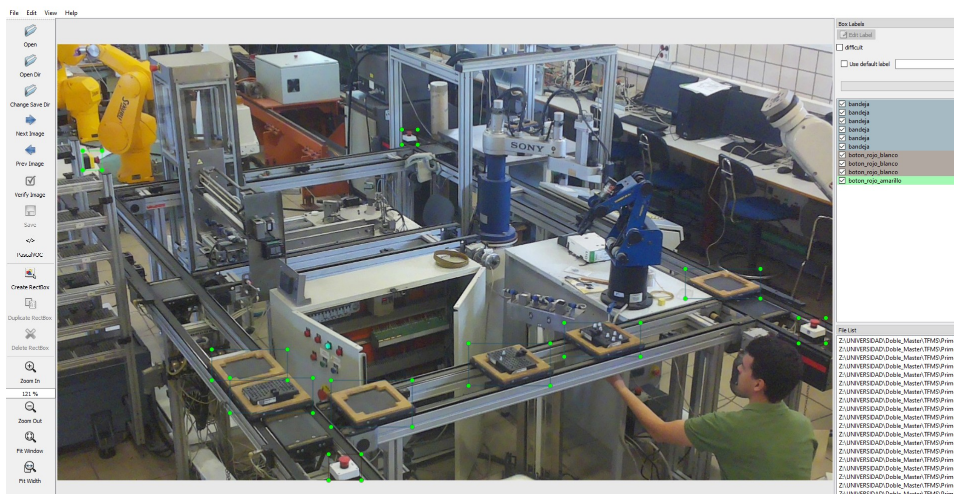

Figura 3.5 Fotograma usado para entrenamiento de red neuronal etiquetada en labelImg.

En el apéndice A.2 se muestra el código para poder pasar la carpeta que tenemos de dataset en formato VOC específico para SSD, a formato YOLO para que se pueda entrenar mediante YOLO.

3.1.3 SSD (Single Shot MultiBox Detector)

El SSD es un algoritmo de detección de objetos que realiza la detección en una sola pasada a través de la red, lo que significa que no necesita una etapa de propuesta de regiones como en métodos como R-CNN. Funciona aplicando una serie de filtros (o convoluciones) para detectar objetos a diferentes escalas y aspectos de la imagen.

Funcionamiento Interno:

- SSD divide la imagen en una cuadrícula de celdas y para cada celda, predice tanto las cajas delimitadoras como las probabilidades de las clases de objetos.
- Utiliza múltiples cajas de diferentes proporciones y escalas para cada ubicación de la cuadrícula.

- Combina predicciones de varias capas de características con diferentes resoluciones para manejar objetos de diferentes tamaños.

Ventajas:

- Rápido y eficiente, adecuado para aplicaciones en tiempo real.
- No requiere una etapa de propuesta de regiones, lo que reduce el tiempo de cómputo.
- Maneja múltiples escalas de objetos gracias a sus cajas de anclaje de diferentes tamaños.

Inconvenientes:

- Puede tener un rendimiento inferior en la detección de objetos muy pequeños en comparación con métodos que utilizan una región de propuestas.
- Requiere un equilibrio entre velocidad y precisión: más capas de características pueden mejorar la precisión pero reducir la velocidad.

En nuestra implementación, se entrena la red neuronal con 20 épocas y con número de batch 8, teniendo unos resultados a primera vista buenos, exceptuando una parte en particular en la que las bandejas no se ven completamente bien por lo que al realizar el seguimiento se complica más de lo que debería, es por ello que a pesar de la implementación de la misma tanto para realizar el entreno con el código del apendice A.4 y el código para su visualización y verificación de resultados en el código del apéndice A.5, se muestra en la figura 3.6, que una de las tres bandejas no se detecta correctamente. Se ha realizado el muestreo del tiempo de inferencia ejecutándolo en el ordenador de alta gama frente a la Raspberry Pi 4, resultando en los rangos de la tabla 3.1.

De aquí se toman las siguientes conclusiones:

- El tipo de red neuronal entrenada en este caso es compatible únicamente para ser usada en tiempo real en el ordenador de alta gama, definiendo tiempo real con una tasa de alimentación de datos hacia un modelo externo de unos 200ms entre recepción y recepción de datos, siendo inviable técnicamente en la Raspberry Pi 4.
- Es posible realizar una red neuronal menos avanzada y con menor coste computacional, para que la Raspberry Pi 4 pueda procesarlo con la CPU en un tiempo razonable.
- Se puede requerir de un acelerador USB para que la Raspberry Pi 4 tenga un tiempo de inferencia mucho menor.

Se opta finalmente por buscar un tipo de red neuronal más sencillo para probar si es posible realizar el proceso con la Raspberry Pi que se tiene actualmente.

3.1.4 YOLO (You Only Look Once)

YOLO es otro algoritmo de detección de objetos en tiempo real que, como su nombre indica, mira la imagen solo una vez para predecir qué objetos hay y dónde se encuentran.

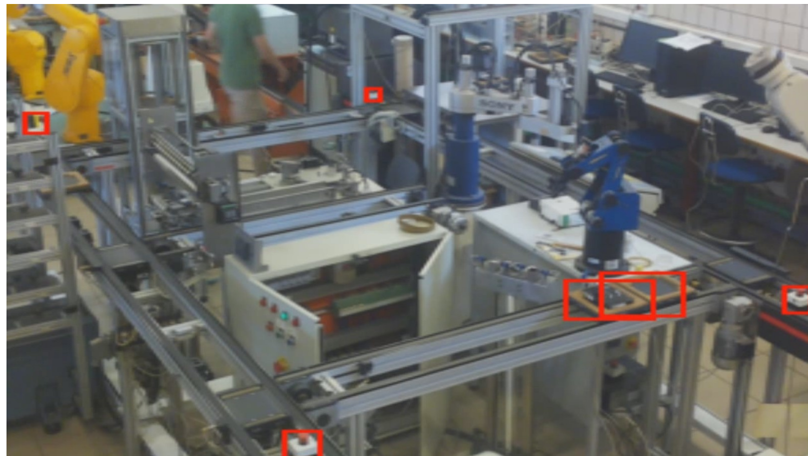

Figura 3.6 Detección fotograma mediante SSD.

Tabla 3.1 Comparación de los tiempos de inferencia media..

Dispositivo	Tiempo de inferencia medio
Ordenador de alta gama	50-55 ms
Raspberry Pi 4	5000-6000 ms

Funcionamiento Interno:

- Divide la imagen en una cuadrícula y para cada celda de la cuadrícula, predice cajas delimitadoras y probabilidades de clase.
- Cada caja delimitadora predice el centro, las dimensiones y la confianza de la detección, que es la probabilidad de que contenga un objeto.
- La red utiliza características de toda la imagen para predecir cada caja delimitadora, lo que le permite capturar el contexto global de la imagen.

Ventajas:

- Extremadamente rápido, permitiendo su uso en aplicaciones de video en tiempo real.
- Predice con un alto grado de precisión gracias a su enfoque global de la imagen.
- Sencillo de entrenar y optimizar debido a su arquitectura de red única.

Inconvenientes:

- La precisión en la detección de objetos pequeños puede ser menor debido a su enfoque global.
- Puede tener dificultades con objetos que aparecen en grupos o que están muy cerca unos de otros.
- La versión original de YOLO puede tener más falsos positivos en comparación con métodos más lentos y precisos.

Procedimiento de Entrenamiento con YOLO

Para entrenar el modelo YOLO con datos personalizados, se siguen los siguientes pasos utilizando el repositorio oficial de Ultralytics para YOLOv5:

1. Clonar el repositorio de GitHub de Ultralytics YOLOv5.
2. Preparar el conjunto de datos personalizado en el formato requerido por YOLOv5.
3. Configurar el archivo de configuración YAML para el entrenamiento, especificando las rutas del conjunto de datos, el número de clases y los nombres de las clases.
4. Ejecutar el script de entrenamiento con los parámetros deseados, como el número de epochs y el tamaño del batch.

Los comands utilizado para iniciar el proceso de entrenamiento con 50 epochs y un tamaño de batch de 32 es el siguiente:

```
python train.py --img 640 --batch 32 --epochs 50 --data custom_dataset.yaml --weights yolov5s.pt
python train.py --img 640 --batch 32 --epochs 50 --data custom_dataset.yaml --weights yolov5n.pt
```

Estos comando debe ejecutarse en la línea de comandos después de activar el entorno correspondiente donde YOLOv5 está instalado. Mostramos a continuación parte de los datos que ofrece al ejecutarse los prompts.

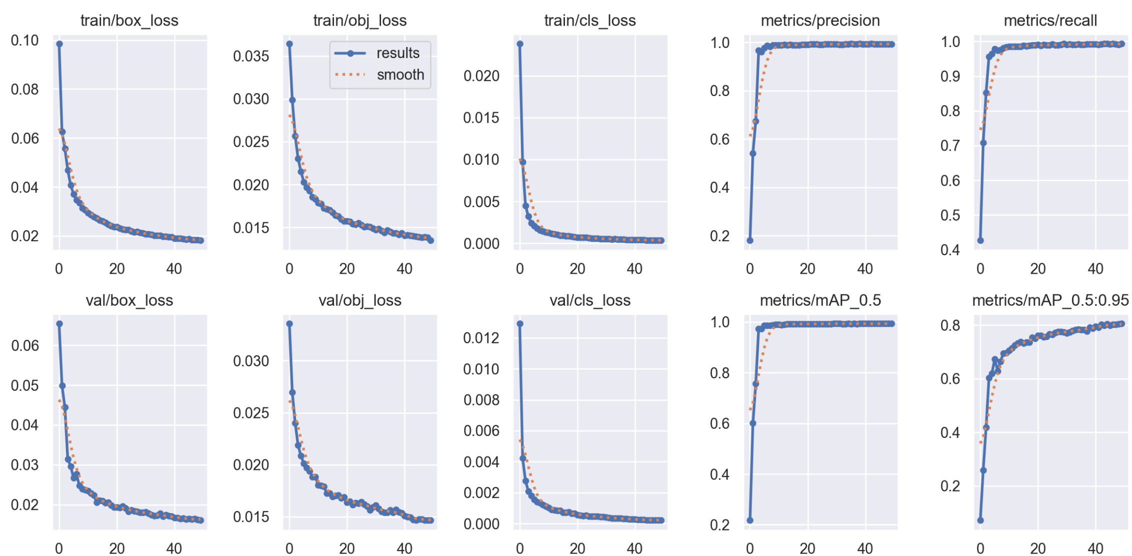

Figura 3.7 Resultados obtenidos del entrenamiento de la red neuronal Yolov5n.

Se comprobado el rango de tiempo de inferencia de la red neuronal pequeña (yolov5s) y la red neuronal muy pequeña (yolov5n), tanto en el ordenador de alta gama como en la Rapsberry Pi 4, y se ha obtenido los siguientes resultados:

Tabla 3.2 Comparación de los tiempos de inferencia medios para diferentes redes neuronales y dispositivos..

Dispositivo	Red Neuronal	Tiempo de inferencia medio
Ordenador de alta gama	YOLOv5s	15-35 ms
Ordenador de alta gama	YOLOv5n	15-30 ms
Raspberry Pi 4	YOLOv5s	1000-1200 ms
Raspberry Pi 4	YOLOv5n	700-800 ms

Respecto a los resultados obtenidos con este método, se comprueba que es mucho más veloz que el anterior y teniendo un buen resultados tanto en el caso más sencillo o visible completamente de las bandejas como en el caso en el que fallaba más el método SSD, por lo que se opta por seguir con el proceso con este tipo de red neuronal, al ser mucho más liviana que la anterior, y teniendo un resultado excelente.

Figura 3.8 Fotograma detección por Yolov5n.

3.2 Tracking de objetos

El tracking de objetos es el siguiente problema al que se enfrenta después de la detección de los mismos objetos en el procesamiento de video y la visión por computadora. Su objetivo es seguir la trayectoria de uno o más objetos en movimiento a través de una serie de marcos de video. Es importante realizar esto ya que se quiere entrenar al gemelo digital con los datos del mundo real y para ello hay que saber en la posición que se encuentra un objeto de manera única en el espacio, por lo que es el siguiente paso en el proceso de posicionar en las imágenes 2D obtenidas.

3.2.1 Conceptos generales

El seguimiento de objetos implica la detección de un objeto en un marco inicial y luego encontrar su posición en los marcos sucesivos. Esta tarea puede ser desafiante debido a cambios en la iluminación, deformaciones del objeto, oclusiones y movimientos rápidos o irregulares. En nuestro caso, el más desafiante es la oclusión en distintos puntos al tener únicamente una cámara

Existen diversas metodologías para realizar el tracking. Algunas se basan en el seguimiento por detección, donde cada marco se procesa para detectar objetos, y luego se utiliza un algoritmo para asociar las detecciones a lo largo del tiempo. Otras técnicas incluyen el seguimiento basado en el modelo, que utiliza un modelo del objeto para guiar el proceso de seguimiento, y el seguimiento basado en el contorno, que sigue la forma del objeto a través de los marcos. En este trabajo se va a optar por implementar el algoritmo de tracking SORT.

3.2.2 SORT

Simple Online and Realtime Tracking (SORT) es un algoritmo para el tracking de objetos en tiempo real que combina métodos de detección de objetos modernos con un filtro de Kalman clásico. El filtro de Kalman es una técnica estadística que estima el estado de un sistema dinámico a partir de una serie de mediciones a lo largo del tiempo, lo que lo hace ideal para predecir la posición de objetos en movimiento cuando las mediciones pueden ser ruidosas o incompletas.

La razón para implementar un filtro de Kalman con SORT es aprovechar su eficiencia computacional y su capacidad para manejar incertidumbres en las mediciones. Cuando se combina con detecciones robustas, SORT puede seguir con eficacia múltiples objetos y proporcionar una identificación consistente a lo largo del tiempo, incluso cuando los objetos se mueven rápidamente o se ocluyen temporalmente.

Para la implementación inicial, partiremos del repositorio proporcionado por RizwanMunawar, que se encuentra disponible en GitHub bajo el nombre `yolov5-object-tracking`. A partir de este código base, tomaremos el archivo `sort.py` y realizaremos las modificaciones pertinentes para adaptarlo a las especificidades de nuestro caso de uso, asegurando que el seguimiento sea óptimo para nuestras condiciones específicas de operación. El código de seguimiento implementado se encuentra en el apéndice A.5, y en el apéndice A.6 se muestran las clases importantes del archivo *sort.py*.

El método SORT (Simple Online and Realtime Tracking) se ha adaptado para el seguimiento robusto de objetos, en particular bandejas, en secuencias de video. Al detectar un objeto, se inicializa una instancia de `KalmanBoxTracker`, que emplea un filtro de Kalman para predecir la posición futura del objeto. Una modificación clave es la capacidad del algoritmo de continuar el seguimiento incluso cuando un objeto es ocluido o desaparece temporalmente de la escena. Esto se logra mediante la evaluación de la estabilidad del movimiento del objeto y, ante eventos de oclusión significativa, utilizando la última velocidad estable conocida para predecir la posición del objeto. Esta aproximación asegura un seguimiento continuo y fluido, vital para aplicaciones que demandan una respuesta en tiempo real. Se muestra en la figura 3.9 un fotograma de la visualizació del vídeo en el que se muestran la detección del cuadro delimitador de la bandeja actual de color azul, la delimitación del trackeo con un cuadro amarillo, y mediante una flecha verde la posición del centro y del centro predicho en este caso. En el caso que mostramos, realizamos un guardado de la posición del centro como del centro predicho para el trackeo de ambas bandejas, con número de ID 2 y ID 5 respectivamente, y tal como se muestran en las figura 3.10, la trayectoria sobre la imagen 2D es prácticamente idéntica sobre las partes en las que no hay oclusión, y se desvía en la parte en la que se oclusiona en las imágenes. Aún con esto, el rastreador y el objeto detectado siguen ambos ligados y no se crea un nuevo rastreador.

Las limitaciones que pueden existir respecto a la modificación realizada son las siguientes:

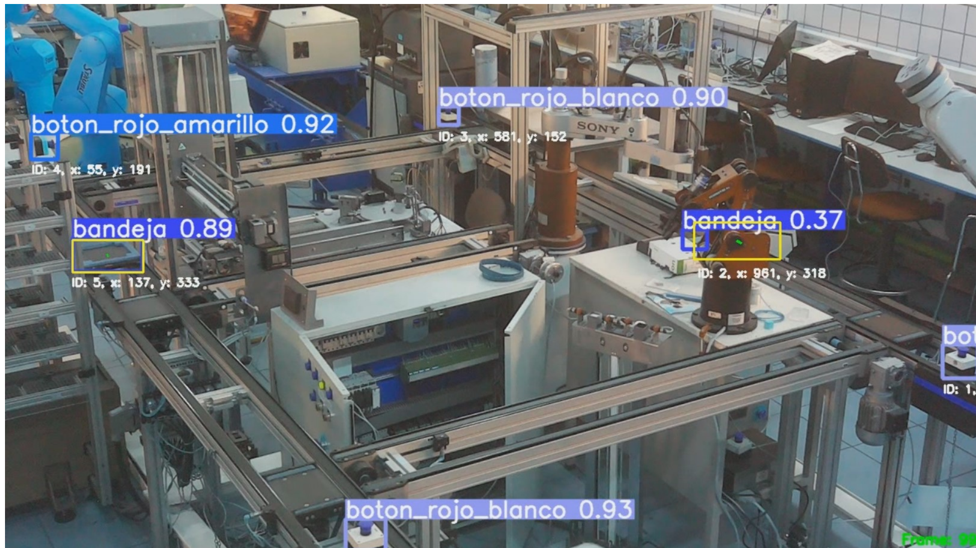

Figura 3.9 Fotograma para seguimiento de las bandejas detectadas.

1. El objeto, en este caso, la bandeja, no puede tener movimientos bruscos en la parte en la que hay oclusión, ya que el rastreador lo sigue buscando en la dirección en la que iba. Es por ello que para subsanar esto, deberían de aplicarse técnicas más sofisticadas de rastreo.

2. Se ha preparado al entorno para que las bandejas no tengan modificaciones durante su trayectoria por el recorrido. Esto es debido a que en la etapa de entrenamiento y obtención de resultados, no ha sido posible realizar experimentos con los brazos robóticos que se emplean en la célula de fabricación, por lo que para realizar este trabajo, se ha simplificado el proceso a una cadena de bandejas que siguen unos carriles, no es realmente una célula de fabricación.

3. Respecto a lo anteriores, todas las bandejas se consideran iguales, tengan un objeto encima de la misma, estén vacías o no, por lo que a la hora de conocer o identificar el objeto que tienen o ha sido alterado no ha sido objeto de este trabajo.

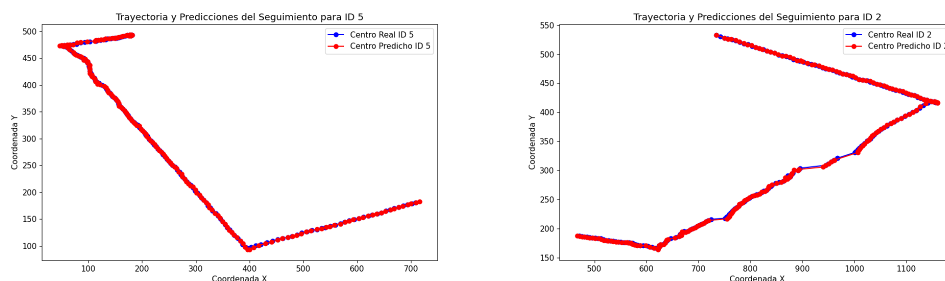

Figura 3.10 Trackeo bandejas ID 2 vs ID 5.

3.3 Introducción al Posicionamiento en el Mundo Real

El reconocimiento y seguimiento de objetos en imágenes bidimensionales (2D) es un área bien establecida en el campo de la visión por computadora. Este proceso generalmente involucra la detección de objetos dentro de una secuencia de imágenes y el seguimiento de su movimiento

y cambios a lo largo del tiempo. Aunque el seguimiento 2D proporciona información valiosa sobre la dinámica de los objetos en el plano de la imagen, carece de una dimensión crucial: la profundidad. Sin esta tercera dimensión, la comprensión completa de la escena y la interacción con objetos en aplicaciones del mundo real está limitada. Por tanto, el último paso tras la identificación y seguimiento de objetos en el plano 2D es la reconstrucción de su posición en el espacio tridimensional (3D).

Convertir la posición 2D de un objeto detectado en una imagen en su ubicación 3D en el mundo real es un problema complejo. Para abordar este problema, se han desarrollado varios métodos que permiten estimar la posición 3D a partir de la información 2D. En este trabajo, exploraremos dos enfoques predominantes para realizar esta conversión: la homografía y la reconstrucción 3D directa.

La homografía es un enfoque que se basa en la relación entre dos imágenes de un mismo plano visto desde diferentes puntos de vista. Mediante el uso de una matriz de homografía, es posible mapear las coordenadas de un objeto en una imagen 2D a su ubicación en un plano del mundo real, asumiendo que el objeto se mantiene en dicho plano. Este método es especialmente útil en aplicaciones como la realidad aumentada y la robótica, donde la comprensión de la posición en un plano es suficiente para la tarea en cuestión.

Por otro lado, el enfoque de reconstrucción 3D intenta estimar la posición y orientación completas de un objeto en el espacio. Utilizando técnicas como 'solvePnP' (Perspective-n-Point), que emplean correspondencias entre puntos 3D del objeto y sus proyecciones 2D en la imagen, es posible inferir la ubicación espacial real del objeto con respecto a la cámara. Este método es crucial en contextos donde la interacción precisa con objetos en un entorno tridimensional es necesaria, como en la manipulación robótica avanzada y la navegación autónoma.

La selección del método adecuado depende en gran medida del contexto de la aplicación y de las limitaciones inherentes a cada enfoque. En las siguientes secciones, detallaremos la implementación de ambos métodos, sus aplicaciones prácticas, y cómo pueden complementarse para obtener una comprensión más rica y detallada del espacio que nos rodea.

3.3.1 Conceptos generales

La homografía es un enfoque que se basa en la relación entre dos imágenes de un mismo plano visto desde diferentes puntos de vista. Mediante el uso de una matriz de homografía, es posible mapear las coordenadas de un objeto en una imagen 2D a su ubicación en un plano del mundo real, asumiendo que el objeto se mantiene en dicho plano. Este método es aplicable en este entorno debido a que en las condiciones trabajadas, todas las bandejas y los carriles se encuentran en el mismo plano horizontal respecto al suelo, por lo que es podemos usar este método para obtener la posición de las bandejas en los carriles. La limitación que tiene es clara, en caso de salir de los carriles, la transformación no sería válida, pero para el trabajo en cuestión, no se ha tenido en cuenta esta posibilidad debido al no uso de otros dispositivos.

Por otro lado, el enfoque de reconstrucción 3D intenta estimar la posición y orientación completas de un objeto en el espacio. Utilizando técnicas como 'solvePnP' (Perspective-n-Point), que emplean correspondencias entre puntos 3D del objeto y sus proyecciones 2D en la imagen, es posible inferir la ubicación espacial real del objeto con respecto a la cámara.

La selección del método adecuado depende en gran medida del contexto de la aplicación y de las limitaciones inherentes a cada enfoque. En las siguientes secciones, detallaremos la implementación de ambos métodos, sus aplicaciones prácticas, y comprobar que ambos métodos funcionan correctamente y estiman la posición de una manera aceptable.

3.3.2 Calibración de la cámara

La calibración de la cámara es un paso crucial para el posicionamiento preciso en 3D. Utiliza patrones conocidos, como tablas Aruco, para determinar los parámetros intrínsecos y extrínsecos de la cámara. Los parámetros intrínsecos incluyen la distancia focal y el punto principal, mientras que los parámetros extrínsecos se relacionan con la orientación y posición de la cámara en el mundo.

Método de calibración

Para calibrar la cámara, se capturaron múltiples imágenes de una tabla Aruco desde diferentes ángulos. Se utilizó la siguiente metodología:

1. Colocación de la tabla Aruco en el campo de visión de la cámara.
2. Captura de imágenes desde diferentes orientaciones y distancias.
3. Detección de los marcadores Aruco en las imágenes y extracción de las esquinas.
4. Uso de las funciones de calibración de OpenCV para estimar los parámetros de la cámara.

El código para la extracción de los párametros intrínsecos de la cámara se pueden ver en el código de prográma del apéndice A.7. En la figura se muestran varias de las imágenes que se han usado para calibrar la cámara. Se ha realizado con una tabla ArUco de 4x5, impresa en un folio A0, con un espacio entre ArUcos de 20mm y un lado de 180mm.

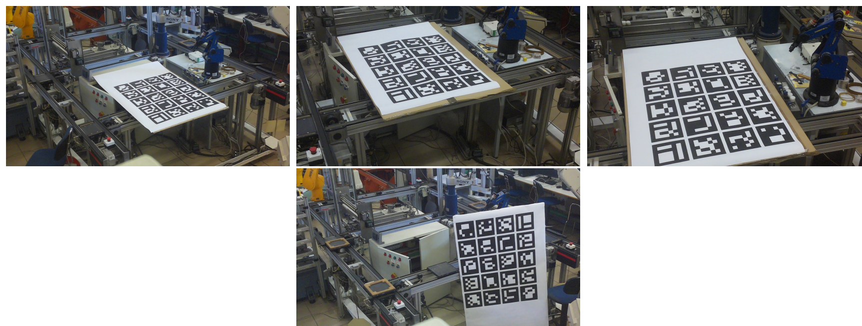

Figura 3.11 Imágenes para la calibración de la Picamera.

3.3.3 Posicionamiento 3D

Una vez que la cámara está calibrada, se puede utilizar la función ‘solvePnP‘ para el posicionamiento 3D. Esta función encuentra la pose de un objeto 3D con respecto a la cámara a partir de puntos correspondientes en una imagen 2D.

Implementación de solvePnP

El método ‘solvePnP‘ se implementó de la siguiente manera:

- Identificación de puntos de referencia en el objeto 3D.
- Correspondencia de estos puntos con sus ubicaciones en la imagen 2D.
- Utilización de ‘solvePnP‘ para calcular la rotación y traslación del objeto.

3.3.4 Posicionamiento mediante homografía

La homografía es una transformación que relaciona dos imágenes de un plano común bajo una perspectiva diferente. La matriz de homografía se calcula utilizando puntos de correspondencia entre el plano y la imagen.

Cálculo de la matriz de homografía

Se calcularon las matrices de homografía utilizando los siguientes pasos:

1. Detección de los marcadores Aruco y de los botones en la imagen y uso de sus coordenadas como puntos de referencia.
2. Aplicación de la función ‘findHomography‘ de OpenCV con estos puntos de referencia.
3. Mapeo de la posición de nuevos objetos detectados en la imagen a sus posiciones en el plano del mundo real.

Las posiciones de los botones blancos y de los Arucos se ha obtenido teniendo como referencia el boton más cercano a la cámara, cuyo recuadro será el mayor de todos, con el eje X que se dirije al segundo botón con mayor tamaño, y el eje Z hacia el plano perpendicular del plano de la célula de fabricación. Se ha tomado así para que los carriles esten alineados con los ejes de coordendas, siendo así más sencillo así su interpretación de movimientos. En la figura 3.12 se muestra el plano de la célula de fabricación con algunos puntos dibujados y en la tabla 3.3 se referencian los puntos. Cabe destacar que finalmente no se ha usado el botón amarillo debido a que no se encuentra en el mismo plano horizontal que las bandejas, por lo que no se le puede aplicar homografía al mismo, aunque por el método SolvePnP sería posible.

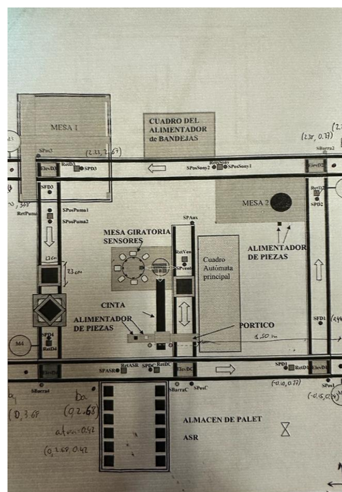

Figura 3.12 Imágenes para la calibración de la Picamera.

Tabla 3.3 Posiciones de referencia para los botones blancos y centros de ArUcos.

ID	Descripción	Posición X (m)	Posición Y (m)
0	ID0_centro ArUco	0.80	1.40
1	ID1_centro ArUco	1.2	1.40
2	boton_referencia	0	0
3	boton1_derecha	2.23	0
4	boton2_fondo	2.23	3.68

4 Resultados

En este capítulo, tras presentar paso a paso los problemas y soluciones requeridas en los apartados anteriores, se va a mostrar el análisis detallado de la posición y el movimiento de una bandeja en una secuencia de 50 frames a 5fps, que ha sido rastreada utilizando tanto el método de homografía como la técnica SolvePnP. Además, se evalúa la velocidad real del objeto, calculada a partir de la variación entre fotogramas y la posición del objeto durante el tiempo.

El código utilizado para resolver todos los apartados se encuentra en el apéndice A.8, y se muestra un fotograma en la figura 4.1.

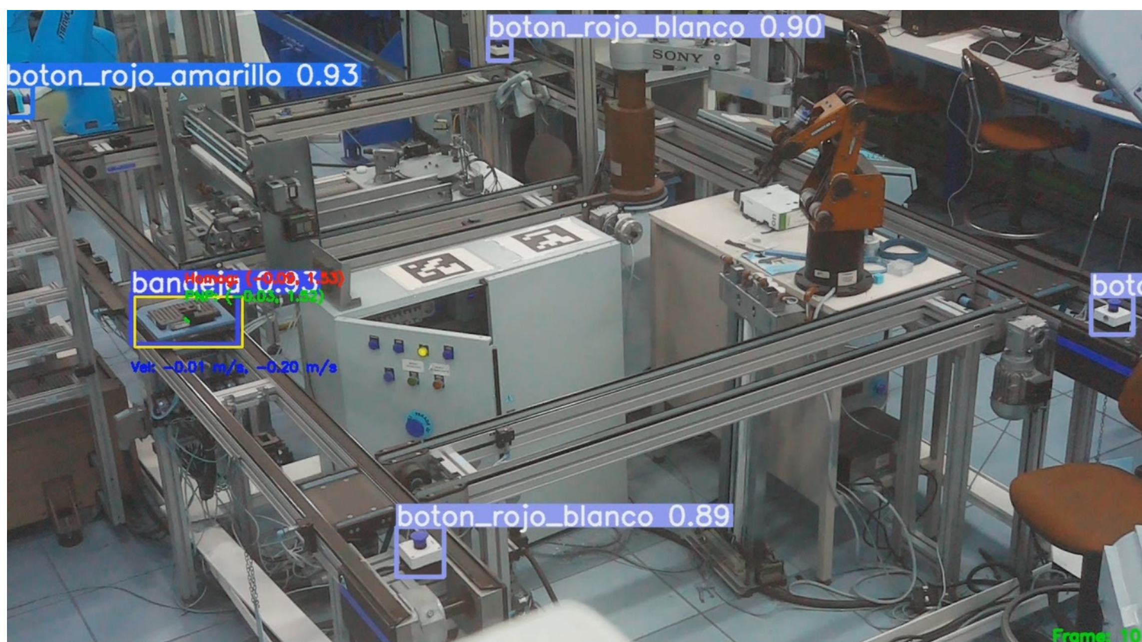

Figura 4.1 Fotograma extraído del vídeo final procesado.

Como se puede ver en la imagen, en la misma se incluyen por un lado, la detección de cada elemento

entrenado mediante machine learning, y en la bandeja, el tracking que se le está realizando a la bandeja, junto con la dirección en una fecha marcada en verde desde el centro de la misma, y sobre puesto, la posición en el plano XY de la bandeja, y debajo del cuadro delimitador, la velocidad tanto en el eje X como en el eje Y. A continuación en las figuras 4.2 y 4.3 se muestran los datos históricos que realiza el objeto trackeado, y como se puede observar, se nota claramente el cambio al llegar al final del carril, ya que pasa de una velocidad negativa en la dirección Y, que significa que se está acercando a la referencia, que como se ha establecido en el botón más cercano a la cámara, es correcto, y pasa a aumentar en la dirección X. Se establece además que la velocidad es prácticamente constante entorno a unos 0.2 m/s, lo cual es bastante acertado.

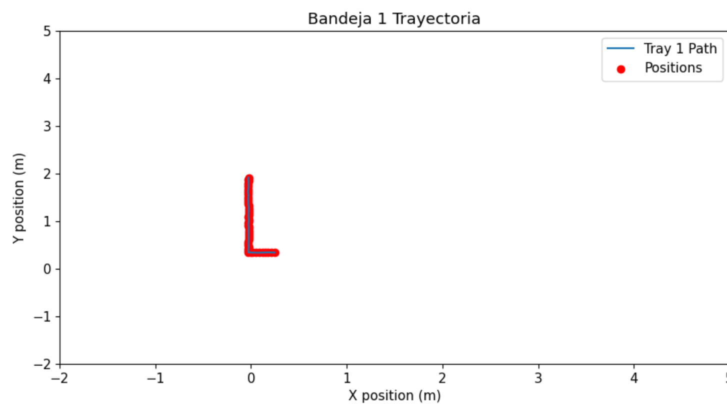

Figura 4.2 Posición bandeja en el plano XY real.

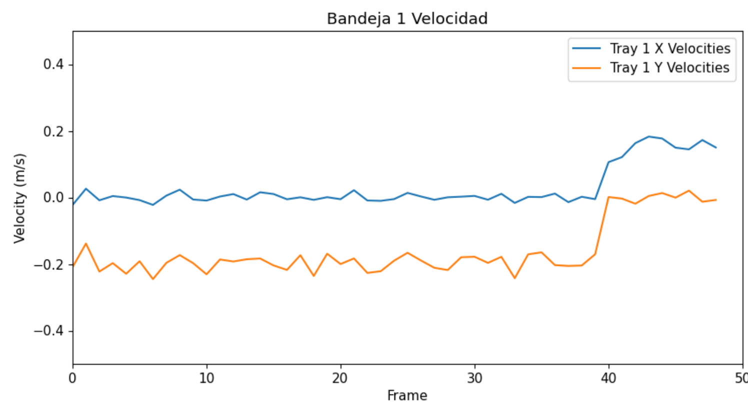

Figura 4.3 Velocidad detectada en la bandeja en cada dirección.

El tiempo de procesado de cada imagen es de alrededor de 100ms en el ordenador de alta gama, por lo que se podría realizar una alimentación de datos a un modelo digital de entorno a 7-9 veces por segundo, lo cual se puede considerar que es bastante rápido y se podría realizar un entrenaiento en tiempo real del mismo.

Los resultados cumplen satisfactoriamente los objetivos iniciales,que eran la detección, rastreo y

posicionamiento en tiempo real, estableciendo unas condiciones de contorno necesarias para el correcto funcionamiento del mismo.

5 Conclusiones

A lo largo de este trabajo, hemos explorado la intersección de la visión por computadora y el seguimiento de objetos, aplicando una combinación de tecnologías emergentes para abordar desafíos complejos. Las conclusiones que hemos alcanzado no solo reflejan los resultados de nuestra investigación, sino que también iluminan el camino a seguir para futuros desarrollos en el campo.

La implementación de redes neuronales para la detección de objetos ha demostrado ser sorprendentemente eficaz, superando las expectativas y ofreciendo una mejora sustancial sobre los métodos de procesamiento digital tradicionales. Este enfoque ha proporcionado una solidez en la detección de elementos específicos, como botones, donde el entrenamiento exhaustivo y la clara visibilidad han favorecido a las redes neuronales sobre la detección basada en Arucos.

Sin embargo, el seguimiento de objetos ha presentado desafíos adicionales, particularmente en presencia de oclusiones. La adopción de técnicas como DeepSORT o la segmentación podrían ofrecer mejoras significativas en estas situaciones. La gestión de las oclusiones, ya sea antes o después de la transferencia de datos al gemelo digital, es crucial, especialmente en escenarios donde las trayectorias de los objetos no se limitan a rutas predefinidas.

La precisión del posicionamiento en el mundo real se ha identificado como dependiente de una nube de puntos más extensa de lo que inicialmente se había considerado. La inclusión de una cámara adicional, permitiendo la inferencia estéreo, emerge como una solución potencial para obtener una localización más exacta de los objetos.

La interacción con el gemelo digital y la incorporación de datos de sensores revelan la importancia de considerar el error inherente en las mediciones basadas en imágenes. Mientras que la homografía ha sido adecuada para el posicionamiento en este estudio, su aplicabilidad se limita en escenarios con variaciones de altura.

Desde una perspectiva de rendimiento, los resultados han sido muy positivos en un entorno de computación de alta capacidad. Sin embargo, la ausencia de una GPU dedicada en sistemas como la Raspberry Pi plantea limitaciones significativas en tiempo real. Una propuesta de mejora es la integración de un acelerador USB para delegar la inferencia de la red neuronal.

El estudio también ha revelado que la altura de la instalación de la cámara es un factor crítico, siendo necesaria una elevación mayor para una cobertura visual completa del área de trabajo. Además, la iluminación ha demostrado tener un impacto considerable en la confianza de la detección, con una reducción notable bajo condiciones de baja luz.

Para mejorar aún más el sistema propuesto, se sugiere la incorporación de múltiples cámaras, un aumento en la densidad de puntos para el posicionamiento 3D, y la utilización de un acelerador USB para mejorar la capacidad de procesamiento de la Raspberry Pi. Además, se debería considerar un filtrado más sofisticado durante las oclusiones, una clasificación detallada de las bandejas según su contenido, y la integración de brazos robóticos y zonas de trabajo en el entrenamiento para detección y posicionamiento. Finalmente, la detección de personas en las áreas de trabajo podría jugar un papel vital en la implementación de medidas de seguridad como las paradas de emergencia.

Este trabajo no solo evidencia los logros alcanzados, sino que también destaca áreas de oportunidad donde la innovación puede continuar mejorando la precisión, la eficiencia y la seguridad en el ámbito de la visión artificial y el seguimiento de objetos.

Apéndice A

Códigos en Python

A.1 Código para obtener fotogramas para dataset

```
1  import cv2
2  import os
3  import time
4
5  def get_settings():
6      # Solicitar al usuario la frecuencia de fotogramas y la duración de la captura
7      framerate_img = int(input("Introduce la frecuencia de fotogramas (fps): "))
8      capture_duration_img = int(input("Introduce la duración de la captura (en segundos): "))
9      # Permitir al usuario elegir entre tres resoluciones predefinidas
10     print("Elige una resolución:")
11     print("1: 1280x720")
12     print("2: 1600x900")
13     print("3: 1920x1080")
14     resolution_choice = input("Introduce 1, 2 o 3: ")
15     # Asignar la resolución basada en la elección del usuario
16     if resolution_choice == "1":
17         capture_width, capture_height = 1280, 720
18     elif resolution_choice == "2":
19         capture_width, capture_height = 1600, 900
20     elif resolution_choice == "3":
21         capture_width, capture_height = 1920, 1080
22     else:
23         # En caso de una elección inválida, usar una resolución por defecto
24         print("Elección no válida. Usando 1280x720 por defecto.")
25         capture_width, capture_height = 1280, 720
26     return framerate_img, capture_duration_img, capture_width, capture_height
27
28 def gstreamer_pipeline(capture_width, capture_height, framerate, display_width, display_height):
29     # Configurar la pipeline de GStreamer para capturar video con los parámetros deseados
30     return (
31         f"libcamerasrc ! video/x-raw, "
32         f"width=(int){capture_width}, "
33         f"height=(int){capture_height}, "
34         f"framerate=(fraction){framerate}/1 ! "
35         f"videoconvert ! videoscale ! "
```

```
            f"video/x-raw, "
            f"width=(int){display_width}, "
            f"height=(int){display_height} ! appsink"
        )

def save_frames(frames, folder_name):
    # Crear un directorio para guardar los fotogramas si no existe
    if not os.path.exists(folder_name):
        os.makedirs(folder_name)
    # Guardar cada fotograma en el directorio especificado
    for idx, frame in enumerate(frames):
        filename = os.path.join(folder_name, f"frame_{str(idx).zfill(4)}.jpg")
        cv2.imwrite(filename, frame)

def main():
    # Obtener configuraciones del usuario
    framerate_img, capture_duration_img, capture_width, capture_height = get_settings()

    while True:
        # Configurar la framerate y dimensiones de visualización
        framerate = framerate_img
        display_width, display_height = capture_width, capture_height

        # Crear y mostrar la pipeline de GStreamer
        pipeline = gstreamer_pipeline(capture_width, capture_height, framerate, display_width,
    display_height)
        print(f"Using pipeline: \n\t{pipeline}\n\n")

        # Iniciar la captura de video con OpenCV
        cap = cv2.VideoCapture(pipeline, cv2.CAP_GSTREAMER)
        if not cap.isOpened():
            print("Failed to open camera.")
            return

        # Crear una ventana para mostrar el video en vivo
        cv2.namedWindow("Camera", cv2.WINDOW_AUTOSIZE)
        print("Hit Q to exit, G to start recording")

        # Inicializar variables de control para la grabación
        recording = False
        frames = []
        start_time = 0

        while True:
            # Leer un fotograma del video
            ret, frame = cap.read()
            if not ret:
                print("Capture read error")
                break

            # Mostrar el fotograma en la ventana
            cv2.imshow("Camera", frame)

            # Capturar la entrada del teclado
            esc = cv2.waitKey(5)
            if esc == ord('q'):
                return
            elif esc == ord('g') and not recording:
                # Iniciar la grabación
                print("Started recording...")
                recording = True
                start_time = time.time()
```

```python
            elif recording and time.time() - start_time >= capture_duration_img:
                # Finalizar la grabación después del tiempo especificado
                print(f"Finished recording, captured {len(frames)} frames.")
                recording = False
                # Guardar los fotogramas grabados
                folder_index = 1
                while os.path.exists(f"/home/pi/Downloads/Primer_TFM-main/datos/datos{
    folder_index}"):
                    folder_index += 1
                folder_name = f"/home/pi/Downloads/Primer_TFM-main/datos/datos{folder_index}"
                save_frames(frames, folder_name)
                print(f"Frames saved in: {folder_name}")
                frames = []
                # Opciones para salir, volver a grabar o cambiar configuraciones
                print("Press Q to exit, R to re-record or C to change settings")
            elif esc == ord('r'):
                # Reiniciar la grabación
                break
            elif esc == ord('c'):
                # Cambiar configuraciones de captura
                cap.release()
                framerate_img, capture_duration_img, capture_width, capture_height = get_settings
    ()
                break

            # Añadir el fotograma a la lista si se está grabando
            if recording:
                frames.append(frame)

        # Liberar recursos y cerrar ventanas
        cap.release()

    cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

A.2 Código para obtener pasar formato VOC a formato YOLO

```python
import os
import shutil
import xml.etree.ElementTree as ET

# Convertir anotaciones VOC en anotaciones YOLO
def convert_voc_annotation_to_yolo(voc_annotation_path, yolo_annotation_path, class_names):
    # Parsear el archivo XML de anotaciones VOC
    tree = ET.parse(voc_annotation_path)
    root = tree.getroot()
    # Obtener las dimensiones de la imagen
    image_width = int(root.find('size/width').text)
    image_height = int(root.find('size/height').text)

    # Lista para guardar las anotaciones en formato YOLO
    yolo_annotations = []
    # Iterar sobre cada objeto en las anotaciones VOC
    for obj in root.findall('object'):
        # Obtener el nombre de la clase y su índice
        class_name = obj.find('name').text
        class_index = class_names.index(class_name)

```

```
22            # Obtener las coordenadas de la caja delimitadora VOC
23            voc_coords = [ float(obj.find('bndbox/xmin').text), float(obj.find('bndbox/ymin').text),
24                           float(obj.find('bndbox/xmax').text), float(obj.find('bndbox/ymax').text)]
25
26            # Convertir las coordenadas VOC a formato YOLO
27            yolo_coords = [(voc_coords[0] + voc_coords[2]) / (2 * image_width),
28                           (voc_coords[1] + voc_coords[3]) / (2 * image_height),
29                           (voc_coords[2] - voc_coords[0]) / image_width,
30                           (voc_coords[3] - voc_coords[1]) / image_height]
31
32            # Añadir la anotación YOLO a la lista
33            yolo_annotations.append(f"{class_index} {' '.join(map(str, yolo_coords))}")
34
35        # Escribir las anotaciones YOLO en el archivo correspondiente
36        with open(yolo_annotation_path, 'w') as f:
37            f.write("\n".join(yolo_annotations))
38
39    # Función para convertir un dataset completo de VOC a YOLO
40    def voc_to_yolo(voc_path, yolo_path, class_names):
41        # Crear la estructura de directorios para YOLO
42        yolo_images_path = os.path.join(yolo_path, "images", "train")
43        yolo_labels_path = os.path.join(yolo_path, "labels", "train")
44        os.makedirs(yolo_images_path, exist_ok=True)
45        os.makedirs(yolo_labels_path, exist_ok=True)
46
47        # Copiar las imágenes de VOC a YOLO
48        voc_images_path = os.path.join(voc_path, "JPEGImages")
49        for image_file in os.listdir(voc_images_path):
50            shutil.copy(os.path.join(voc_images_path, image_file), os.path.join(yolo_images_path,
       image_file))
51
52        # Convertir las anotaciones VOC a formato YOLO
53        voc_annotations_path = os.path.join(voc_path, "Annotations")
54        for xml_file in os.listdir(voc_annotations_path):
55            voc_annotation_path = os.path.join(voc_annotations_path, xml_file)
56            yolo_annotation_path = os.path.join(yolo_labels_path, os.path.splitext(xml_file)[0] + '.
       txt')
57            convert_voc_annotation_to_yolo(voc_annotation_path, yolo_annotation_path, class_names)
58
59    # Rutas de los directorios VOC y YOLO
60    voc_path = "C:/Users/Ceravedi/Desktop/UNIVERSIDAD/Doble_Master/TFMS/Primer TFM/
           GitHub_1_TFM/Primer_TFM-main_12_10_2023/datos/dataset_VOC"
61    yolo_path = "C:/Users/Ceravedi/Desktop/UNIVERSIDAD/Doble_Master/TFMS/Primer TFM/
           GitHub_1_TFM/Primer_TFM-main_12_10_2023/datos/dataset_YOLO"
62    # Nombres de las clases en el dataset
63    class_names = ["bandeja", "boton_rojo_blanco", "boton_rojo_amarillo"]
64
65    # Ejecutar la conversión del dataset VOC a formato YOLO
66    voc_to_yolo(voc_path, yolo_path, class_names)
```

A.3 Código para entrenamiento mediante SSD

```
1 import os
2 import torch
3 from torchvision.models.detection import ssd300_vgg16
4 from torchvision.transforms import functional as F
5 import xml.etree.ElementTree as ET
6 from PIL import Image
7 from tqdm import tqdm
8 from sklearn.model_selection import train_test_split
```

```python
from torchvision.transforms import ToTensor, Resize
from torch.optim.lr_scheduler import ReduceLROnPlateau
from torch.utils.data import Subset

# Diccionario para mapear nombres de clases a índices
class_to_index = {"bandeja": 1, "boton_rojo_blanco": 2, "boton_rojo_amarillo": 3}

# Clase personalizada para el conjunto de datos
class CustomVOCDataset(torch.utils.data.Dataset):
    def __len__(self):
        # Devuelve el número total de imágenes en el conjunto de datos
        return len(self.imgs)

    def __init__(self, root, transforms=None):
        # Inicializa el conjunto de datos
        self.root = root
        self.transforms = transforms

        # Lista todas las imágenes y anotaciones
        all_imgs = sorted(os.listdir(os.path.join(root, "JPEGImages")))
        all_annotations = sorted(os.listdir(os.path.join(root, "Annotations")))

        # Conserva solo las imágenes para las que tienes anotaciones
        img_names = set([os.path.splitext(img)[0] for img in all_imgs])
        ann_names = set([os.path.splitext(ann)[0] for ann in all_annotations])

        # Intersección de nombres para asegurarse de que ambos, imagen y anotación, existen
        common_files = list(img_names & ann_names)

        # Guarda los nombres de archivos de las imágenes y las anotaciones
        self.imgs = [f"{file}.jpg" for file in sorted(common_files)]
        self.annotations = [f"{file}.xml" for file in sorted(common_files)]

    def __getitem__(self, idx):
        # Obtiene un elemento del conjunto de datos
        img_path = os.path.join(self.root, "JPEGImages", self.imgs[idx])
        annotation_path = os.path.join(self.root, "Annotations", self.annotations[idx])
        img = Image.open(img_path).convert("RGB")

        # Parsea la anotación XML para obtener la información de los bounding boxes
        tree = ET.parse(annotation_path)
        root = tree.getroot()
        boxes = []
        labels = []
        for obj in root.findall("object"):
            label = obj.find("name").text
            bbox = obj.find("bndbox")
            xmin = int(bbox.find("xmin").text)
            ymin = int(bbox.find("ymin").text)
            xmax = int(bbox.find("xmax").text)
            ymax = int(bbox.find("ymax").text)
            boxes.append([xmin, ymin, xmax, ymax])

            # Convierte las etiquetas en índices
            labels.append(class_to_index[label])

        # Verificación para manejar imágenes sin bounding boxes
        if len(boxes) == 0:
            boxes = torch.zeros((0, 4), dtype=torch.float32)
        else:
            boxes = torch.as_tensor(boxes, dtype=torch.float32)

```

```
71          # Actualiza las etiquetas con los índices
72          labels = torch.tensor(labels, dtype=torch.int64)
73
74          # Crea el diccionario objetivo con información de los bounding boxes
75          target = {
76              "boxes": boxes,
77              "labels": labels,
78              "image_id": torch.tensor([idx]),
79              "area": (boxes[:, 3] - boxes[:, 1]) * (boxes[:, 2] - boxes[:, 0]),
80              "iscrowd": torch.zeros((len(boxes),), dtype=torch.int64)
81          }
82
83          # Aplica transformaciones si se han proporcionado
84          if self.transforms:
85              img, target = self.transforms(img, target)
86
87          return img, target
88
89
90  def compute_val_loss(model, val_loader):
91      """
92      Calcular la pérdida media de validación.
93
94      Parámetros:
95      - model: El modelo entrenado.
96      - val_loader: DataLoader para el conjunto de validación.
97
98      Devuelve:
99      - avg_loss: Pérdida media de validación.
100     """
101     model.eval()  # Cambia el modelo al modo de evaluación
102     total_loss = 0.0
103     with torch.no_grad():  # Desactiva el cálculo de gradientes durante la evaluación
104         for images, targets_list in val_loader:
105             images = [image.to(device) for image in images]
106             targets_list = [{k: v.to(device) for k, v in target.items()} for target in
        targets_list]
107
108             # Cambia el modelo al modo de entrenamiento solo para calcular las pérdidas
109             model.train()
110             loss_dict = model(images, targets_list)
111             losses = sum(loss for loss in loss_dict.values())
112             total_loss += losses.item()
113             model.eval()  # Vuelve al modo de evaluación
114
115     avg_loss = total_loss / len(val_loader.dataset)  # Divide por el número total de imágenes
116     return avg_loss
117
118 def transform_with_bbox_fixed(img, target):
119     """
120     Transformación con ajuste de bounding boxes.
121
122     Parámetros:
123     - img: La imagen a transformar.
124     - target: El objetivo que contiene las cajas delimitadoras.
125
126     Devuelve:
127     - img: La imagen transformada.
128     - target: El objetivo con las cajas delimitadoras ajustadas.
129     """
130     # Dimensiones originales
131     old_width, old_height = img.size
```

```
132
133     # Redimensiona la imagen
134     img = F.resize(img, (300, 300))
135
136     # Factores de escala
137     x_scale = 300.0 / old_width
138     y_scale = 300.0 / old_height
139
140     # Ajusta las bounding boxes
141     boxes = target["boxes"].float()  # Convierte a tipo float
142     boxes[:, [0, 2]] *= x_scale  # Ajusta las coordenadas x
143     boxes[:, [1, 3]] *= y_scale  # Ajusta las coordenadas y
144     target["boxes"] = boxes
145
146     # Convierte la imagen a tensor
147     img = F.to_tensor(img)
148
149     return img, target
150
151 def collate_fn(batch):
152     """
153     Función personalizada para agrupar datos en el DataLoader.
154
155     Parámetros:
156     - batch: Un lote de datos.
157
158     Devuelve:
159     - Un lote de imágenes y objetivos procesados.
160     """
161     images, targets = zip(*batch)
162     resize = Resize((300, 300))
163
164     # Convierte las imágenes de PIL a tensores y las redimensiona a 300x300
165     images = [ToTensor()(resize(image)) if not isinstance(image, torch.Tensor) else image for
          image in images]
166     images = torch.stack(images, 0)
167     return images, targets
168
169
170 # Solicitar al usuario el número X
171 X = "20"  # Asumimos en este caso datos20
172
173 # Cargar conjunto de datos personalizado
174 dataset = CustomVOCDataset(
175     root=f"C:/Users/Ceravedi/Desktop/UNIVERSIDAD/Doble_Master/TFMS/Primer TFM/
          GitHub_1_TFM/Primer_TFM-main_12_10_2023/datos/dataset_VOC",
176     transforms=transform_with_bbox_fixed)
177
178 # Dividir el conjunto de datos en entrenamiento y validación
179
180 indices = list(range(len(dataset)))
181 train_indices, val_indices = train_test_split(indices, test_size=0.2, shuffle=True)
182
183 # Crea subconjuntos basados en estos índices
184 train_data = Subset(dataset, train_indices)
185 val_data = Subset(dataset, val_indices)
186 train_loader = torch.utils.data.DataLoader(train_data, batch_size=32, shuffle=True, drop_last=
          True,
187                                            collate_fn=collate_fn)
188 val_loader = torch.utils.data.DataLoader(val_data, batch_size=32, shuffle=False, drop_last=True,
          collate_fn=collate_fn)
189
```

```
190 # Verificar CUDA
191 os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
192 os.environ["CUDA_VISIBLE_DEVICES"] = "0" # Asume que tu GTX 1070 es el dispositivo 0.
193
194 if torch.cuda.is_available():
195     print("CUDA está disponible.")
196     print(f"Dispositivo CUDA actual: {torch.cuda.get_device_name(0)}")
197 else:
198     print("CUDA no está disponible.")
199
200 # Modelo
201 device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
202 model = ssd300_vgg16(weights=False) # Sin pesos preentrenados
203 model.to(device)
204
205 # Entrenamiento
206 params = [p for p in model.parameters() if p.requires_grad]
207 optimizer = torch.optim.SGD(params, lr=0.001, momentum=0.9, weight_decay=0.001)
208 scheduler = ReduceLROnPlateau(optimizer, 'min', patience=5, verbose=True)
209 num_epochs = 30
210
211 # Añadir barra de progreso para las épocas
212 for epoch in tqdm(range(num_epochs), desc="Epochs"):
213     model.train()
214     total_train_loss = 0.0
215
216     # Añadir barra de progreso para los batches
217     for images, targets_list in tqdm(train_loader, desc="Batches", leave=False):
218         images = [image.to(device) for image in images]
219         targets_list = [{k: v.to(device) for k, v in target.items()} for target in targets_list]
220
221         optimizer.zero_grad()
222         loss_dict = model(images, targets_list)
223         losses = sum(loss for loss in loss_dict.values())
224         total_train_loss += losses.item()
225         losses.backward()
226
227         # Clipping de gradientes
228         torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1)
229         optimizer.step()
230
231         # Mostrar la pérdida del modelo después de cada batch
232         print(f"Loss: {losses.item()}")
233
234     avg_train_loss = total_train_loss / len(train_loader.dataset)
235     print(f"Avg Training Loss after epoch {epoch + 1}: {avg_train_loss}")
236
237     val_loss = compute_val_loss(model, val_loader)
238     scheduler.step(val_loss)
239
240     print(f"Validation Loss after epoch {epoch + 1}: {val_loss}")
241
242
243 torch.save(model.state_dict(), "ssd300_vgg_model_bandeja_boton1.pth")
```

A.4 Código para verificación por vídeo mediante SSD

```
1 import torch
2 import torchvision.transforms as T
3 from torchvision.models.detection import ssd300_vgg16
```

```
4  from PIL import Image, ImageDraw
5  import os
6  import cv2
7  import numpy as np
8  import time
9
10 # Cargar el modelo
11 device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
12 model = ssd300_vgg16(pretrained=False)  # No cargar pesos preentrenados
13 model.load_state_dict(torch.load("C:/Users/Ceravedi/Desktop/UNIVERSIDAD/Doble_Master/TFMS/
       Primer TFM/GitHub_1_TFM/Primer_TFM-main_12_10_2023/red_neuronal/
       ssd300_vgg_model_bandeja_boton1.pth"))
14 model = model.to(device).eval()  # Cambiar modelo a modo evaluación
15
16 # Transformaciones: Asegurarse de que sean las mismas que se usaron durante el entrenamiento
17 transform = T.Compose([
18     T.Resize((300, 300)),
19     T.ToTensor()
20 ])
21
22 # Solicitar al usuario la carpeta que desea leer
23 base_folder = "C:/Users/Ceravedi/Desktop/UNIVERSIDAD/Doble_Master/TFMS/Primer TFM/
       GitHub_1_TFM/Primer_TFM-main_12_10_2023/datos/"
24 folder_name = input("Introduce el nombre de la carpeta (por ejemplo, datos1): ")
25 full_folder_path = os.path.join(base_folder, folder_name)
26
27 # Preparar la escritura del video
28 video_name = f"video_{folder_name}.mp4"
29 fourcc = cv2.VideoWriter_fourcc(*'mp4v')
30 out = cv2.VideoWriter(video_name, fourcc, 15.0, (1280, 720))
31
32 # Iterar sobre cada imagen en la carpeta
33 for image_file in sorted(os.listdir(full_folder_path)):
34     image_path = os.path.join(full_folder_path, image_file)
35     img = Image.open(image_path).convert("RGB")
36
37     img_transformed_tensor = transform(img)
38     img_tensor = img_transformed_tensor.unsqueeze(0).to(device)  # Añadir dimensión de batch y
       enviar al dispositivo
39
40     # Medir tiempo inicial
41     start_time = time.time()
42
43     # Realizar detección
44     with torch.no_grad():
45         prediction = model(img_tensor)
46
47     # Medir tiempo final
48     elapsed_time = time.time() - start_time
49
50     # Analizar resultados
51     threshold = 0.5
52     detected_boxes = [box for box, score in zip(prediction[0]['boxes'], prediction[0]['scores'])
       if score > threshold]
53
54     # Dibujar recuadros en la imagen
55     img_transformed = T.ToPILImage()(img_transformed_tensor)  # Convertir tensor a imagen PIL
56     draw = ImageDraw.Draw(img_transformed)
57     for box in detected_boxes:
58         draw.rectangle(box.tolist(), outline="red", width=2)  # Dibuja recuadro delimitador en
       rojo
59
```

```
        # Convertir la imagen PIL a formato OpenCV y ajustar el tamaño
        open_cv_image = cv2.cvtColor(np.array(img_transformed), cv2.COLOR_RGB2BGR)
        resized_image = cv2.resize(open_cv_image, (1280, 720))
        out.write(resized_image)

        # Imprimir tiempo de procesamiento
        print(f"Imagen: {image_file} - Tiempo de procesamiento: {elapsed_time:.4f} segundos")

out.release()
print(f"Video generado con éxito: {video_name}")
```

A.5 Código para implementación SORT

```
import torch
import os
import sys
sys.path.append("Z:/yolov5-object-tracking-main")
os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"
import cv2
from PIL import Image
import time
import numpy as np
from sort import *


# --- INICIALIZACIÓN DEL MODELO YOLOv5 ---
# Cargar el modelo YOLOv5 personalizado
model_path = "Z:/yolov5-master/runs/train/exp7/weights/best.pt"
model = torch.hub.load('Z:/yolov5-master', 'custom', path=model_path, source='local')
model.eval()
# --- FIN DE LA INICIALIZACIÓN ---

# Initialize SORT tracker
mot_tracker = Sort(min_hits=30, iou_threshold=0.01)

# --- CONFIGURACIÓN DE ENTRADA Y SALIDA ---
# Solicitar al usuario la carpeta que desea leer
base_folder = "Z:/UNIVERSIDAD/Doble_Master/TFMS/Primer_TFM/GitHub_1_TFM/Primer_TFM-
    main_12_10_2023/datos/"
folder_name = input("Introduce el nombre de la carpeta (por ejemplo, datos1): ")
full_folder_path = os.path.join(base_folder, folder_name)

# Tomar una imagen de muestra para obtener sus dimensiones
img_sample = cv2.imread(os.path.join(full_folder_path, os.listdir(full_folder_path)[0]))
height, width, _ = img_sample.shape
print(f"Dimensiones de la imagen original: Ancho = {width}, Alto = {height}")

# Preparar la escritura del video
video_name = f"video_yolov5nanoKALMAN{folder_name}.mp4"
fourcc = cv2.VideoWriter_fourcc(*'mp4v')
out = cv2.VideoWriter(video_name, fourcc, 15.0, (width, height))
# --- FIN DE LA CONFIGURACIÓN ---

# --- PROCESAMIENTO DE IMÁGENES Y DETECCIÓN ---

# Iterar sólo sobre las imágenes en la carpeta
# --- PROCESAMIENTO DE IMÁGENES Y DETECCIÓN ---
first_iteration = True
# Iterar sobre todas las imágenes en la carpeta especificada con extensión '.jpg'
for image_file in sorted([f for f in os.listdir(full_folder_path) if f.endswith('.jpg')]):
```

```
47
48        image_path = os.path.join( full_folder_path , image_file )
49        img = Image.open(image_path).convert("RGB")
50
51        start_time = time.time()
52        # Realizar detección de objetos en la imagen usando el modelo YOLOv5
53        results = model(img, size=640)
54        # Extrae la imagen con los cuadros delimitadores de ‘ results ‘
55        img_with_boxes = results.render() [0].copy()
56
57        # Filtrar las detecciones que superen el umbral de confianza
58        threshold=0.75
59        detected_boxes = [np.concatenate((box.cpu().numpy(), [cls.item()], [0, 0, 0, 0])) for box,
          score, cls in
60                           zip( results.pred [0][:, :4], results.pred [0][:, 4], results.pred [0][:, 5]) if
          score > threshold]
61
62        if first_iteration :
63            first_iteration = False
64
65        mot_tracker.update(np.array(detected_boxes))
66        trackers_act = mot_tracker.getTrackers()
67
68        for tracker in trackers_act :
69            track = tracker.get_state() [0]
70            x1, y1, x2, y2, class_detected , vx, vy, cambio_escala = map(int, track)
71            track_id = tracker.id
72            # Sólo imprimir si class_detected es 0
73            if class_detected == 0:
74                print(f"Track ID: {track_id} − Cambio de escala: {cambio_escala}")
75
76            # Calculate midpoint of the bounding box
77            center_x = (x1 + x2) // 2
78            center_y = (y1 + y2) // 2
79
80            # Draw the ID and center coordinates on the image
81            label = f"ID: {track_id}, x: {center_x}, y: {center_y}"
82            cv2.putText(img_with_boxes, label, (x1, y2 + 20), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
          (255, 255, 255), 2)
83
84            if class_detected == 0: # if it’s of class 0
85                # Compute predicted center based on velocity
86                pred_center_x = center_x + vx
87                pred_center_y = center_y + vy
88
89                # Draw an arrow from current center to predicted center
90                cv2.arrowedLine(img_with_boxes, (center_x, center_y), (pred_center_x, pred_center_y),
          (0, 255, 0), 2)
91
92                # Compute the predicted bounding box corners based on the predicted center
93                width = x2 − x1
94                height = y2 − y1
95                pred_x1 = pred_center_x − width // 2
96                pred_y1 = pred_center_y − height // 2
97                pred_x2 = pred_center_x + width // 2
98                pred_y2 = pred_center_y + height // 2
99
100               # Draw the predicted bounding box
101               cv2.rectangle(img_with_boxes, (pred_x1, pred_y1), (pred_x2, pred_y2), (0, 255, 255),
          2)
102
103
```

```
        elapsed_time = time.time() - start_time


        # Imprimir el tiempo que tardó en procesar esta imagen
        print(f"Imagen: {image_file} - Tiempo de procesamiento y trackeo: {elapsed_time:.4f} segundos"
         )

        #imprime frame en la imagen
        # Extraer el número de frame del nombre del archivo
        frame_number = int(image_file.split("_")[1].split(".")[0])  # Esto extraerá el número XXXX
         de frame_XXXX.jpg

        # Mostrar el número de frame en la esquina inferior derecha de la imagen
        text = f"Frame: {frame_number}"
        font = cv2.FONT_HERSHEY_SIMPLEX
        font_scale = 0.6
        font_thickness = 2
        color = (0, 255, 0)  # Verde
        text_size = cv2.getTextSize(text, font, font_scale, font_thickness)[0]

        # La posición (x, y) es la esquina inferior derecha del texto.
        # Así que, para que el texto aparezca en la esquina inferior derecha de la imagen:
        x = img_with_boxes.shape[1] - text_size[0] - 10  # 10 pixels de margen desde el borde derecho
        y = img_with_boxes.shape[0] - 10  # 10 pixels de margen desde el borde inferior

        cv2.putText(img_with_boxes, text, (x, y), font, font_scale, color, font_thickness, lineType=
         cv2.LINE_AA)

        #fin imprimir en la imagen


        # Guardar la imagen con las detecciones y flechas dibujadas en una ubicación temporal
        save_path = os.path.join(full_folder_path, "temp_img_with_boxes.jpg")
        cv2.imwrite(save_path, img_with_boxes)
        # Añade la imagen modificada al video de salida
        out.write(img_with_boxes)

# Una vez que todas las imágenes han sido procesadas y añadidas al video, finalizar y guardar el
     video
os.remove(save_path)  # Eliminar la imagen temporal
out.release()  # Cerrar y guardar el archivo de video
print(f"Video generado con éxito: {video_name}")
```

A.6 Clases SORT y KalmanBoxTracker

```
def linear_assignment(cost_matrix):
    try:
        import lap #linear assignment problem solver
        _, x, y = lap.lapjv(cost_matrix, extend_cost = True)
        return np.array([[y[i], i] for i in x if i>=0])
    except ImportError:
        from scipy.optimize import linear_sum_assignment
        x,y = linear_sum_assignment(cost_matrix)
        return np.array(list(zip(x,y)))


"""From SORT: Computes IOU between two boxes in the form [x1,y1,x2,y2]"""
def iou_batch(bb_test, bb_gt):

    bb_gt = np.expand_dims(bb_gt, 0)
```

```
16      bb_test = np.expand_dims(bb_test, 1)
17
18      xx1 = np.maximum(bb_test [...,0], bb_gt [..., 0])
19      yy1 = np.maximum(bb_test [..., 1], bb_gt [..., 1])
20      xx2 = np.minimum(bb_test [..., 2], bb_gt [..., 2])
21      yy2 = np.minimum(bb_test [..., 3], bb_gt [..., 3])
22      w = np.maximum(0., xx2 - xx1)
23      h = np.maximum(0., yy2 - yy1)
24      wh = w * h
25      o = wh / ((bb_test [..., 2] - bb_test [..., 0]) * (bb_test [..., 3] - bb_test [..., 1])
26      + (bb_gt [..., 2] - bb_gt [..., 0]) * (bb_gt [..., 3] - bb_gt [..., 1]) - wh)
27      return (o)
28
29
30  """Takes a bounding box in the form [x1,y1,x2,y2] and returns z in the form [x,y,s,r] where x,y is
        the center of the box and s is the scale/area and r is the aspect ratio """
31  def convert_bbox_to_z(bbox):
32      w = bbox[2] - bbox[0]
33      h = bbox[3] - bbox[1]
34      x = bbox[0] + w/2.
35      y = bbox[1] + h/2.
36      s = w * h
37      #scale is just area
38      r = w / float(h)
39      return np.array([x, y, s, r]).reshape((4, 1))
40
41
42  """Takes a bounding box in the centre form [x,y,s,r] and returns it in the form
43      [x1,y1,x2,y2] where x1,y1 is the top left and x2,y2 is the bottom right """
44  def convert_x_to_bbox(x, score=None):
45      w = np.sqrt(x[2] * x[3])
46      h = x[2] / w
47      if(score==None):
48          return np.array([x[0]-w/2.,x[1]-h/2., x[0]+w/2.,x[1]+h/2.]).reshape((1,4))
49      else:
50          return np.array([x[0]-w/2.,x[1]-h/2., x[0]+w/2.,x[1]+h/2., score]).reshape((1,5))
51
52  """This class represents the internal state of individual tracked objects observed as bbox."""
53  class KalmanBoxTracker(object):
54
55      count = 0
56
57      def velocidad_crucero(self):
58          # Analizar los últimos 7 a 10 estados para comprobar la estabilidad
59          if len(self.state_history) >= 3:  # Asegurarse de que hay al menos 3 estados para comparar

60              # Obtener los últimos 3 estados
61              recent_states = np.array(self.state_history[-3:])
62
63              # Calcular la varianza para los estados de interés (índices del 2 al 6)
64              self.variances = np.var(recent_states[:, 2:7], axis=0)
65
66              #ver por pantalla
67              # Nombres de las variables correspondientes a las varianzas
68              variance_names = ['s', 'r', 'x_dot', 'y_dot', 's_dot']
69              for name, var in zip(variance_names, self.variances):
70                  # Asegurarse de que var es un escalar antes de formatearlo
71                  if isinstance(var, np.ndarray):
72                      # Si var es un array con un solo elemento, obtenemos ese elemento.
73                      var = var.item()
74                  #print(f"{name} variance = {var:.6f}")
75
```

```
76            # Definir los umbrales para cada estado de interés
77            # Los umbrales deben ser definidos como una lista o un array, no como asignaciones de
     variable dentro de una lista
78            # thresholds = [threshold_s, threshold_r, threshold_x_dot, threshold_y_dot,
     threshold_s_dot] # Estos son valores hipotéticos
79            thresholds = [10000,0.0005, 0.04, 0.04, 700] # Estos son valores hipotéticos
80
81            # Comprobar si la varianza es menor que el umbral para cada estado
82            if all(var < thresh for var, thresh in zip(self.variances, thresholds)):
83                # Fijar el estado a los valores estables si es necesario
84                self.corrected_state_history = recent_states[-1]
85                print("Actualizo velocidad de crucero a la actual")
86
87
88
89    def __init__(self, bbox):
90        """
91         Initialize a tracker using initial bounding box
92
93        Parameter 'bbox' must have 'detected class' int number at the -1 position.
94        """
95        self.kf = KalmanFilter(dim_x=7, dim_z=4)
96        self.kf.F = np.array
          ([[1,0,0,0,1,0,0],[0,1,0,0,0,1,0],[0,0,1,0,0,0,1],[0,0,0,1,0,0,0],[0,0,0,0,1,0,0],[0,0,0,0,0,1,0],[0,0,0,0,0,0,1]])

97        self.kf.H = np.array     ([[1,0,0,0,0,0,0],[0,1,0,0,0,0,0],[0,0,1,0,0,0,0],[0,0,0,1,0,0,0]])
98
99        self.kf.R[2:,2:] *= 10. # R: Covariance matrix of measurement noise (set to high for noisy
      inputs -> more 'inertia' of boxes')
100       self.kf.P[4:,4:] *= 1000. #give high uncertainty to the unobservable initial velocities
101       self.kf.P *= 10.
102       self.kf.Q[-1,-1] *= 0.5 # Q: Covariance matrix of process noise (set to high for
      erratically moving things)
103       self.kf.Q[4:,4:] *= 0.5
104
105       self.kf.x[:4] = convert_bbox_to_z(bbox) # STATE VECTOR
106       self.time_since_update = 0
107       self.id = KalmanBoxTracker.count
108       KalmanBoxTracker.count += 1
109       self.history = []
110       self.hits = 0
111       self.hit_streak = 0
112       self.age = 0
113       self.centroidarr = []
114
115       self.variances = [0.0]*5
116       self.state_history = []
117       self.corrected_state_history = [0.0]*7
118       self.stable_state = [] #valor de velocidad de crucero
119
120       CX = (bbox[0]+bbox[2])//2
121       CY = (bbox[1]+bbox[3])//2
122       self.centroidarr.append((CX,CY))
123       self.flag_control_automatico = 0
124
125
126       #keep yolov5 detected class information
127       self.detclass = bbox[4]
128
129   def update(self, bbox):
130       """
131       Updates the state vector with observed bbox
```

```
        """
        self.time_since_update = 0
        self.history = []
        self.hits += 1
        self.hit_streak += 1
        self.kf.update(convert_bbox_to_z(bbox))
        self.state_history.append(self.kf.x.copy())
        self.detclass = bbox[4]
        if self.flag_control_automatico == 0:
            CX = (bbox[0]+bbox[2])//2
            CY = (bbox[1]+bbox[3])//2
            self.centroidarr.append((CX,CY))
        elif self.flag_control_automatico == 1:
            # Asegúrate de que corrected_state_history es lo suficientemente grande
            if len(self.corrected_state_history) >= 7:

                CX = self.centroidarr[-1][0] + self.corrected_state_history[4]  # Asumiendo que el índice 4 es la velocidad en x
                CY = self.centroidarr[-1][1] + self.corrected_state_history[5]  # Asumiendo que el índice 5 es la velocidad en y
                self.centroidarr.append((CX, CY))



    def predict(self):
        """
        Advances the state vector and returns the predicted bounding box estimate
        """
        if((self.kf.x[6]+self.kf.x[2])<=0): #no quiero que se haga infinitamente pequeño
            self.kf.x[6] *= 0.0

        if self.detclass == 0:
            self.velocidad_crucero()

            # Definir los umbrales para cada estado de interés
            thresholds = [10000, 0.0005, 0.04, 0.04, 700]  # Estos son valores hipotéticos
            # Suponiendo que self.variances es una lista
            variances_print = self.variances

            # Convertir la lista en un array de NumPy
            variances_array = np.array(variances_print)

            # Aplanar el array (en este caso, es redundante ya que el array ya está en 1D)
            variances_flat = variances_array.flatten()

            # Imprimir las varianzas aplanadas
            print("Variances:", ' '.join(f"{var:.6f}" for var in variances_flat))

            print(f"Thresholds: {thresholds}")
            if self.variances[1] > thresholds[1] and self.kf.x[6]<-40:

                self.flag_control_automatico = 1
                # Actualizar el estado interno del filtro de Kalman con el estado corregido
                self.kf.x[2:7] = np.array(self.corrected_state_history[2:7]).reshape(-1, 1)
                self.kf.x[0] = self.centroidarr[-1][0] + self.kf.x[4]
                self.kf.x[1] = self.centroidarr[-1][1] + self.kf.x[5]
            else:
                self.flag_control_automatico =0

            self.kf.predict()

        # Imprimir los valores
```

```
192
193             self.age += 1
194             if(self.time_since_update>0):
195                 self.hit_streak = 0
196             self.time_since_update += 1
197             self.history.append(convert_x_to_bbox(self.kf.x))
198
199
200
201             # bbox=self.history[-1]
202             # CX = (bbox[0]+bbox[2])/2
203             # CY = (bbox[1]+bbox[3])/2
204             # self.centroidarr.append((CX,CY))
205
206             self.state_history.append(self.kf.x.copy())
207
208
209             return self.history[-1]
210
211
212         def get_state(self):
213             """
214             Returns the current bounding box estimate
215             # test
216             arr1 = np.array([[1,2,3,4]])
217             arr2 = np.array([0])
218             arr3 = np.expand_dims(arr2, 0)
219             np.concatenate((arr1,arr3), axis=1)
220             """
221             arr_detclass = np.expand_dims(np.array([self.detclass]), 0)
222
223             arr_u_dot = np.expand_dims(self.kf.x[4],0)
224             arr_v_dot = np.expand_dims(self.kf.x[5],0)
225             arr_s_dot = np.expand_dims(self.kf.x[6],0)
226
227             return np.concatenate((convert_x_to_bbox(self.kf.x), arr_detclass, arr_u_dot, arr_v_dot,
          arr_s_dot), axis=1)
228
229     def associate_detections_to_trackers(detections, trackers, iou_threshold = 0.01):
230         """
231         Assigns detections to tracked object (both represented as bounding boxes)
232         Returns 3 lists of
233         1. matches,
234         2. unmatched_detections
235         3. unmatched_trackers
236         """
237         if len(detections) == 0:
238             return np.empty((0, 2), dtype=int), np.empty((0, 5), dtype=int), np.arange(len(trackers))
239
240         if(len(trackers)==0):
241             return np.empty((0,2),dtype=int), np.arange(len(detections)), np.empty((0,5),dtype=int)
242
243         iou_matrix = iou_batch(detections, trackers)
244
245         if min(iou_matrix.shape) > 0:
246             a = (iou_matrix > iou_threshold).astype(np.int32)
247             if a.sum(1).max() == 1 and a.sum(0).max() ==1:
248                 matched_indices = np.stack(np.where(a), axis=1)
249             else:
250                 matched_indices = linear_assignment(-iou_matrix)
251         else:
252             matched_indices = np.empty(shape=(0,2))
```

```
253
254     unmatched_detections = []
255     for d, det in enumerate( detections ):
256         if(d not in matched_indices [:,0]) :
257             unmatched_detections.append(d)
258
259     unmatched_trackers = []
260     for t, trk in enumerate( trackers ):
261         if(t not in matched_indices [:,1]) :
262             unmatched_trackers.append(t)
263
264     # filter out matched with low IOU
265     matches = []
266     for m in matched_indices:
267         if(iou_matrix[m[0], m[1]]<iou_threshold):
268             unmatched_detections.append(m[0])
269             unmatched_trackers.append(m[1])
270         else:
271             matches.append(m.reshape(1,2))
272
273     if(len(matches)==0):
274         matches = np.empty((0,2), dtype=int)
275     else:
276         matches = np.concatenate(matches, axis=0)
277
278     return matches, np.array(unmatched_detections), np.array(unmatched_trackers)
279
280
281 class Sort(object):
282     def __init__(self, max_age=40, min_hits=6, iou_threshold =0.1):
283         """
284         Parameters for SORT
285         """
286         self.max_age = max_age
287         self.min_hits = min_hits
288         self.iou_threshold = iou_threshold
289         self.trackers = []
290         self.frame_count = 0
291     def getTrackers(self,):
292         return self.trackers
293
294     def update(self, dets= np.empty((0,6))):
295         """
296         Parameters:
297         'dets' - a numpy array of detection in the format [[x1, y1, x2, y2, score], [x1,y1,x2,y2,
        score ],...]
298
299         Ensure to call this method even frame has no detections. (pass np.empty((0,5)))
300
301         Returns a similar array, where the last column is object ID (replacing confidence score)
302
303         NOTE: The number of objects returned may differ from the number of objects provided.
304         """
305         self.frame_count += 1
306
307         # Get predicted locations from existing trackers
308         trks = np.zeros((len(self.trackers), 6))
309         to_del = []
310         ret = []
311         for t, trk in enumerate(trks):
312             pos = self.trackers[t].predict() [0]
313             trk[:] = [pos[0], pos[1], pos[2], pos[3], 0, 0]
```

```
314             if np.any(np.isnan(pos)):
315                 to_del.append(t)
316         trks = np.ma.compress_rows(np.ma.masked_invalid(trks))
317         for t in reversed(to_del):
318             self.trackers.pop(t)
319         matched, unmatched_dets, unmatched_trks = associate_detections_to_trackers(dets, trks,
        self.iou_threshold)
320
321         # Update matched trackers with assigned detections
322         for m in matched:
323             self.trackers[m[1]].update(dets[m[0], :])
324
325         # Create and initialize new trackers for unmatched detections
326         for i in unmatched_dets:
327             trk = KalmanBoxTracker(np.hstack((dets[i,:], np.array([0]))))
328             #trk = KalmanBoxTracker(np.hstack(dets[i,:])
329             self.trackers.append(trk)
330
331         i = len(self.trackers)
332         for trk in reversed(self.trackers):
333             d = trk.get_state()[0]
334             if (trk.time_since_update < 1) and (trk.hit_streak >= self.min_hits or self.
        frame_count <= self.min_hits):
335                 ret.append(np.concatenate((d, [trk.id+1])).reshape(1,-1)) #+1'd because MOT
        benchmark requires positive value
336             i -= 1
337             #remove dead tracklet
338             if(trk.time_since_update > self.max_age):
339                 self.trackers.pop(i)
340         if(len(ret) > 0):
341             return np.concatenate(ret)
342         return np.empty((0,6))
```

A.7 Detector de tabla ArUco para calibración de la cámara

```
1 from typing import List, Any, Union, Sequence
2 import cv2
3 import os
4 import glob
5 import numpy as np
6 from cv2 import Mat
7 from cv2.aruco import GridBoard
8 from numpy import ndarray, dtype, floating, generic, signedinteger
9 from numpy._typing import _32Bit
10
11 # print(cv2.getBuildInformation())
12
13
14 # Definir la ruta al directorio que contiene las imágenes
15  folder_path = 'Z:/UNIVERSIDAD/Doble_Master/TFMS/Primer_TFM/GitHub_1_TFM/Primer_TFM-
        main_12_10_2023/calibracion/datos46/'
16
17 camera_matrix_0 = np.load('Z:/UNIVERSIDAD/Doble_Master/TFMS/Primer_TFM/GitHub_1_TFM/
        Primer_TFM-main_12_10_2023/calibracion/inicia_camara/camera_matrix.npy')
18  dist_coeffs_0 = np.load('Z:/UNIVERSIDAD/Doble_Master/TFMS/Primer_TFM/GitHub_1_TFM/
        Primer_TFM-main_12_10_2023/calibracion/inicia_camara/dist_coeffs.npy')
19
20 # Carga de la imagen.( o de las imagenes)
21 # inputImage = cv2.imread('C:/Users/Ceravedi/Downloads/frame_0000.jpg')
22 # Calibracion de la camara
```

```
23
24 # Parámetros del tablero Aruco
25 ids = np.array([962, 548, 438, 337, 222, 974, 769, 401, 916, 827, 412, 602, 409, 254, 117, 589,
       886, 227, 268, 927])
26 markersX = 4  # Número de marcadores en dirección X
27 markersY = 5  # Número de marcadores en dirección Y
28 markerLength = 0.18  # Tamaño del marcador en metros (100 mm)
29 markerSeparation = 0.02  # Separación entre marcadores en metros (20 mm)
30
31
32 def calibrate_and_save_parameters():
33     # Define the aruco dictionary and charuco board
34     ids = np.array([962, 548, 438, 337, 222, 974, 769, 401, 916, 827, 412, 602, 409, 254, 117,
       589, 886, 227, 268, 927])
35     dictionary = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_ARUCO_ORIGINAL)
36     board: GridBoard = cv2.aruco.GridBoard((markersX, markersY), markerSeparation, markerLength,
       dictionary, ids)
37     params = cv2.aruco.DetectorParameters()
38     detector = cv2.aruco.ArucoDetector(dictionary, params)
39
40     # Load PNG images from folder
41     image_files = [os.path.join(folder_path, f) for f in os.listdir(folder_path) if f.endswith(".
       jpg")]
42     #image_files.sort()  # Ensure files are in order
43
44     all_corners = np.array([])
45     all_ids = np.array([])
46     num_detected_markers = []
47
48     for image_file in image_files:
49         image = cv2.imread(image_file)
50         img_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
51         #cv2.imshow("obama", img_gray)
52         #cv2.waitKey(0)
53         #cv2.destroyAllWindows()
54         corners, ids, rejected = detector.detectMarkers(img_gray)
55         corners, ids, rejected, recovered= detector.refineDetectedMarkers(img_gray,board,corners,ids,
       rejected,None,None)
56         if np.size(all_corners) == 0:
57             all_corners = corners
58             all_ids = ids
59         else:
60             all_corners = np.append(all_corners, corners, axis=0)
61             all_ids = np.append(all_ids, ids, axis=0)
62         num_detected_markers.append(len(ids))
63
64     num_detected_markers = np.array(num_detected_markers)
65
66     # Ensure the types and shapes are correct before calling the calibration function
67     # Finalmente, llama a la función de calibración
68     retval, camera_matrix, dist_coeffs, rvecs, tvecs = cv2.aruco.calibrateCameraAruco(all_corners,
       all_ids, num_detected_markers, board, [1280,720], camera_matrix_0, dist_coeffs_0)
69     #Save calibration data
70     np.save('camera_matrix.npy', camera_matrix)
71     np.save('dist_coeffs.npy', dist_coeffs)
72
73     # Iterate through displaying all the images
74     for image_file in image_files:
75         image = cv2.imread(image_file)
76          dist_coeffs_restricted = np.array([dist_coeffs[0][0], dist_coeffs[0][1], dist_coeffs
       [0][2], dist_coeffs[0][3], 60])
77         undistorted_image = cv2.undistort(image, camera_matrix, dist_coeffs_restricted)
```

```
            cv2.imshow('Undistorted Image', undistorted_image)
            cv2.waitKey(0)
            cv2.destroyAllWindows()

        print("Matriz de la cámara:")
        print(camera_matrix)

        print("\nCoeficientes de distorsión:")
        print(dist_coeffs)
calibrate_and_save_parameters()
```

A.8 Código completo para detección,rastreo y posicionamiento de bandejas en video

```
import torch
import os
import sys
sys.path.append("Z:/yolov5-object-tracking-main")
os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"
import cv2
from PIL import Image
import time
import numpy as np
from sort import *
import matplotlib.pyplot as plt

# Diccionarios para almacenar la última posición conocida y la velocidad de cada rastreador
last_known_positions = {}
velocities_history = {}
time_between_frames = 0.2  # 5 fps equivalen a 0.2 segundos entre cada frame
# Inicializa un diccionario para almacenar las posiciones y velocidades de las bandejas
bandejas_data = {}


object_points = np.array([
    [0.80, 1.40, 0],  # ID0_centro
    [1.2, 1.40, 0],  # ID1_centro
    [0, 0, 0],  # boton0_real
    [2.23, 0, 0],  # boton1_real
    [2.23, 3.68, 0],  # boton2_real
])

img_points=np.array([])

def get_world_position(u, v, camera_matrix, rvec, tvec, s=None):
    # Generamos el vector m
    uv = np.array([[u, v, 1]], dtype=float).T
    # Obtenemos R a partir de rvec
    R, _ = cv2.Rodrigues(rvec)
    Inv_R = np.linalg.inv(R)

    # Parte izquierda m*A^(-1)*R^(-1)
    Izda = Inv_R.dot(np.linalg.inv(camera_matrix).dot(uv))
    # Parte derecha
    Drch = Inv_R.dot(tvec)
    # Calculamos S porque sabemos Z = 0
    if s is None:
        s = 0 + Drch[2][0] / Izda[2][0]

    XYZ = Inv_R.dot(s * np.linalg.inv(camera_matrix).dot(uv) - tvec)
```

```
47
48        return XYZ.flatten()
49
50 def estimate_position_homography(u, v, H):
51        uv_homog = np.array([u, v, 1]).reshape(-1, 1)
52        world_pos_homog = np.dot(H, uv_homog)
53        world_pos_homog /= world_pos_homog[2] # Normaliza para hacer la coordenada homogénea igual a
            1
54        return world_pos_homog[:2].flatten()
55
56
57 # Define la función que calcula la matriz de homografía
58 def calculate_homography(detected_image_points, detected_object_points):
59        # Asegúrate de que los puntos estén en el formato correcto
60        detected_image_points_hom = np.array(detected_image_points, dtype='float32')
61        detected_object_points_hom = np.array(detected_object_points, dtype='float32')[:, :2]
62        H, _ = cv2.findHomography(detected_image_points_hom, detected_object_points_hom)
63        return H
64
65 # Define la función que utiliza SolvePnP para obtener rvec y tvec
66 def calculate_solvepnp(detected_image_points, detected_object_points, camera_matrix, dist_coeffs):
67        detected_image_points_np = np.array(detected_image_points, dtype='float32')
68        object_points_np = np.array(detected_object_points, dtype='float32')
69        success, rvec, tvec = cv2.solvePnP(object_points_np, detected_image_points_np, camera_matrix,
            dist_coeffs)
70        return success, rvec, tvec
71
72 # Suponiendo que tus archivos se llamen 'camera_matrix.npy' y 'dist_coeffs.npy'
73 camera_matrix = np.load('camera_matrix.npy')
74 dist_coeffs = np.load('dist_coeffs.npy')
75 dictionary = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_6X6_1000)
76 params = cv2.aruco.DetectorParameters()
77 detector = cv2.aruco.ArucoDetector(dictionary, params)
78
79 # --- INICIALIZACIÓN DEL MODELO YOLOv5 ---
80 # Cargar el modelo YOLOv5 personalizado
81 model_path = "Z:/yolov5-master/runs/train/exp7/weights/best.pt"
82 model = torch.hub.load('Z:/yolov5-master', 'custom', path=model_path, source='local')
83 model.eval()
84 # --- FIN DE LA INICIALIZACIÓN ---
85
86 # Initialize SORT tracker
87 mot_tracker = Sort(min_hits=30, iou_threshold=0.01)
88
89 # --- CONFIGURACIÓN DE ENTRADA Y SALIDA ---
90 # Solicitar al usuario la carpeta que desea leer
91 base_folder = "Z:/UNIVERSIDAD/Doble_Master/TFMS/Primer_TFM/GitHub_1_TFM/Primer_TFM-
        main_12_10_2023/datos/"
92 folder_name = input("Introduce el nombre de la carpeta (por ejemplo, datos1): ")
93 full_folder_path = os.path.join(base_folder, folder_name)
94
95 # Tomar una imagen de muestra para obtener sus dimensiones
96 img_sample = cv2.imread(os.path.join(full_folder_path, os.listdir(full_folder_path)[0]))
97
98 # Preparar la escritura del video
99 video_name = f"video_yolov5nanoKALMAN{folder_name}.mp4"
100 fourcc = cv2.VideoWriter_fourcc(*'mp4v')
101 out = cv2.VideoWriter(video_name, fourcc, 5.0, (1280, 720))
102 # --- FIN DE LA CONFIGURACIÓN ---
103
104 # --- PROCESAMIENTO DE IMÁGENES Y DETECCIÓN ---
105
```

```
106 # Iterar sólo sobre las imágenes en la carpeta
107 # --- PROCESAMIENTO DE IMÁGENES Y DETECCIÓN ---
108   first_iteration = True
109 # Iterar sobre todas las imágenes en la carpeta especificada con extensión '.jpg'
110 for image_file in sorted([f for f in os.listdir(full_folder_path) if f.endswith('.jpg')]):
111 
112     start_time = time.time()
113 
114     image_path = os.path.join(full_folder_path, image_file)
115     img = Image.open(image_path).convert("RGB")
116     img2= cv2.imread(image_path)
117     img_gray = cv2.cvtColor(img2,cv2.COLOR_BGR2GRAY)
118     corners, ids, rejected = detector.detectMarkers(img_gray)
119     # Realizar detección de objetos en la imagen usando el modelo YOLOv5
120     results = model(img, size=640)
121     # Extrae la imagen con los cuadros delimitadores de `results`
122     img_with_boxes = results.render()[0].copy()
123 
124     # Filtrar las detecciones que superen el umbral de confianza
125     threshold=0.7
126     detected_boxes = [np.concatenate((box.cpu().numpy(), [cls.item()], [0, 0, 0, 0])) for box,
      score, cls in zip(results.pred[0][:, :4], results.pred[0][:, 4], results.pred[0][:, 5]) if
      score > threshold]
127 
128     # Variables para los puntos detectados
129     image_points_botones_blancos = []  # Solo para botones blancos
130     detected_image_points = []
131     detected_object_points = []
132 
133     # Añade los puntos de ArUco si están presentes
134     if ids is not None:  # Verifica que se hayan detectado ArUcos
135         for i, corner in enumerate(corners):
136             id = ids[i, 0]  # El ID del ArUco detectado
137             # Asegúrate de que el ID está dentro de los límites de tus ArUcos definidos (0, 1, 2)
138             if id < 2:
139                 center_x = np.mean(corner[0][:, 0])
140                 center_y = np.mean(corner[0][:, 1])
141                 detected_image_points.append([center_x, center_y])
142                 detected_object_points.append(
143                     object_points[id])  # Añade el object_point correspondiente al ID del ArUco
144 
145     # Divide las detecciones en bandejas, botones blancos y botón amarillo
146     for det in detected_boxes:
147         x1, y1, x2, y2, cls,_,_,_,_ = det
148         center_x = (x1 + x2) / 2
149         center_y = (y1 + y2) / 2
150         if cls == 1:  # Botones blancos
151             area = (x2 - x1) * (y2 - y1)
152             image_points_botones_blancos.append([center_x, center_y, area])
153 
154     # Ordena los botones blancos por área (de mayor a menor)
155     image_points_botones_blancos.sort(key=lambda x: x[2], reverse=True)
156 
157     # Ahora añade los botones blancos ordenados a las listas de puntos detectados
158     for i, point in enumerate(image_points_botones_blancos):
159         if i < 3:  # Solo tienes tres botones blancos en object_points
160             detected_image_points.append(point[:2])  # Añade solo las coordenadas x, y
161             detected_object_points.append(object_points[2 + i])  # Añade los object points de los
      botones blancos
162 
163     # Utiliza solvePnP para encontrar la pose de la cámara
164     if len(detected_image_points) >= 4:
```

```
165             detected_image_points_np = np.array(detected_image_points, dtype='float32')
166             object_points_np = np.array(object_points, dtype='float32')
167 
168             success, rotation_vector, translation_vector = cv2.solvePnP(
169                 object_points_np, detected_image_points_np, camera_matrix, dist_coeffs
170             )
171 
172             # Convert lists to NumPy arrays
173             detected_image_points = np.array(detected_image_points, dtype=np.float32)
174             detected_object_points = np.array( detected_object_points , dtype=np.float32)
175 
176             # If the points are in shape (n, 2) reshape them to (n, 1, 2)
177             if len(detected_image_points.shape) == 2:
178                 detected_image_points = detected_image_points.reshape(-1, 1, 2)
179             if len( detected_object_points .shape) == 2:
180                 detected_object_points_hom = detected_object_points [:, :2].reshape(-1, 1, 2)
181 
182             # Now, find the homography
183             H, _ = cv2.findHomography(detected_image_points, detected_object_points_hom)
184 
185         mot_tracker.update(np.array(detected_boxes))
186         trackers_act = mot_tracker.getTrackers()
187 
188 
189         for tracker in trackers_act:
190             track = tracker.get_state()[0]
191             x1, y1, x2, y2, class_detected, vx, vy, cambio_escala = map(int, track)
192             track_id = tracker.id
193 
194             # Calculate midpoint of the bounding box
195             center_x = (x1 + x2) // 2
196             center_y = (y1 + y2) // 2
197 
198             if class_detected == 0: # if it's of class 0
199                 # Compute predicted center based on velocity
200                 pred_center_x = center_x + vx
201                 pred_center_y = center_y + vy
202 
203                 # Draw an arrow from current center to predicted center
204                 cv2.arrowedLine(img_with_boxes, (center_x, center_y), (pred_center_x, pred_center_y),
          (0, 255, 0), 2)
205 
206                 # Compute the predicted bounding box corners based on the predicted center
207                 width = x2 - x1
208                 height = y2 - y1
209                 pred_x1 = pred_center_x - width // 2
210                 pred_y1 = pred_center_y - height // 2
211                 pred_x2 = pred_center_x + width // 2
212                 pred_y2 = pred_center_y + height // 2
213 
214                 # Draw the predicted bounding box
215                 cv2.rectangle(img_with_boxes, (pred_x1, pred_y1), (pred_x2, pred_y2), (0, 255, 255),
          2)
216 
217                 # Usando solvePnP para obtener la posición en el mundo real
218                 world_position_pnp = get_world_position(center_x, center_y, camera_matrix,
          rotation_vector, translation_vector )
219 
220                 # Usando homografía para obtener la posición estimada en el mundo real
221                 world_position_homography = estimate_position_homography(center_x, center_y, H)
222 
223                 # Dibuja las coordenadas del mundo real en la imagen (SolvePnP)
```

```
224             cv2.putText(img_with_boxes, f"PNP: ({world_position_pnp[0]:.2f}, {world_position_pnp
        [1]:.2f})",
225                         (int(center_x), int(center_y) - 20), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
        (0, 255, 0), 2)
226 
227             # Dibuja las coordenadas del mundo real estimadas por homografía
228             cv2.putText(img_with_boxes,
229                         f"Homog: ({world_position_homography[0]:.2f}, {world_position_homography
        [1]:.2f})",
230                         (int(center_x), int(center_y) - 40), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
        (0, 0, 255), 2)
231             # Si tenemos una posición previa, podemos calcular la velocidad
232             # Verificar si ya se tiene una posición previa para este rastreador
233             if track_id not in last_known_positions:
234                 bandejas_data[track_id] = {
235                     'positions': [world_position_pnp],
236                     'velocities': []  # No hay velocidad para la primera posición
237                 }
238                 last_known_positions[track_id] = world_position_pnp
239                 velocities_history[track_id] = [np.array([0, 0, 0])]  # Velocidad inicial cero
240                 continue  # Pasar al siguiente rastreador, no hay velocidad previa para comparar
241 
242                 # Obtener la última posición conocida
243             last_position = last_known_positions[track_id]
244             # Calcular la diferencia de posición
245             position_difference = world_position_pnp - last_position
246             # Calcular la velocidad (diferencia de posición entre tiempo entre fotogramas)
247             speed = position_difference / time_between_frames
248             # Almacenar la velocidad en el historial de velocidades
249             velocities_history[track_id].append(speed)
250             # Almacenar la posición actual como la última posición conocida
251             last_known_positions[track_id] = world_position_pnp
252 
253             # Dibujar la velocidad actual en la imagen
254             speed_text = f"Vel: {speed[0]:.2f} m/s, {speed[1]:.2f} m/s"
255             cv2.putText(img_with_boxes, speed_text, (x1, y2 + 30), cv2.
        FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
256             # Obtén la última posición conocida y calcula la velocidad
257             last_position = bandejas_data[track_id]['positions'][-1]
258             velocity = (world_position_pnp - last_position) / time_between_frames  # Esto asume
        que 'world_position_pnp' y 'last_position' son np.array
259             bandejas_data[track_id]['positions'].append(world_position_pnp)
260             bandejas_data[track_id]['velocities'].append(velocity)
261 
262         elapsed_time = time.time() - start_time
263         # Imprimir el tiempo que tardó en procesar esta imagen
264         print(f"Imagen: {image_file} - Tiempo de procesamiento y trackeo: {elapsed_time:.4f} segundos"
        )
265 
266         #imprime frame en la imagen
267         try:
268             frame_number = int(image_file.split("_")[1].split(".")[0])
269         except ValueError:
270             print(f"Skipping file with unexpected name format: {image_file}")
271             continue
272 
273         # Mostrar el número de frame en la esquina inferior derecha de la imagen
274         text = f"Frame: {frame_number}"
275         font = cv2.FONT_HERSHEY_SIMPLEX
276         font_scale = 0.6
277         font_thickness = 2
278         color = (0, 255, 0)  # Verde
```

```
279         text_size = cv2.getTextSize(text, font, font_scale, font_thickness)[0]
280 
281         # La posición (x, y) es la esquina inferior derecha del texto.
282         # Así que, para que el texto aparezca en la esquina inferior derecha de la imagen:
283         x = img_with_boxes.shape[1] - text_size[0] - 10 # 10 pixels de margen desde el borde derecho
284         y = img_with_boxes.shape[0] - 10 # 10 pixels de margen desde el borde inferior
285         first_iteration = False
286         cv2.putText(img_with_boxes, text, (x, y), font, font_scale, color, font_thickness, lineType=
            cv2.LINE_AA)
287 
288         #fin imprimir en la imagen
289 
290 
291         # Guardar la imagen con las detecciones y flechas dibujadas en una ubicación temporal
292         save_path = os.path.join(full_folder_path, "temp_img_with_boxes.jpg")
293         cv2.imwrite(save_path, img_with_boxes)
294         # Añade la imagen modificada al video de salida
295         out.write(img_with_boxes)
296 
297 # Una vez que todas las imágenes han sido procesadas y añadidas al video, finalizar y guardar el
        video
298 os.remove(save_path) # Eliminar la imagen temporal
299 out.release() # Cerrar y guardar el archivo de video
300 # Al final del procesamiento, puedes exportar el diccionario de velocidades a un archivo para
        MATLAB
301 
302 print(f"Video generado con éxito: {video_name}")
303 
304 
305 # Configurar los límites de los ejes en metros (ajusta estos valores según sea necesario)
306 x_limits = (-2, 5) # Rango de ejemplo para el eje X
307 y_limits = (-2, 5) # Rango de ejemplo para el eje Y
308 frame_limits =(0,100)
309 velocity_limits = (-0.5, 0.5) # Rango de ejemplo para las velocidades
310 
311 # Después de recopilar todos los datos, genera gráficos para cada bandeja rastreada
312 for track_id, data in bandejas_data.items():
313     positions = np.array(data['positions'])
314     velocities = np.array(data['velocities'])
315 
316     # Gráfico de la posición de las bandejas
317     plt.figure(figsize=(10, 5))
318     plt.plot(positions[:, 0], positions[:, 1], label=f'Tray {track_id} Path')
319     plt.scatter(positions[:, 0], positions[:, 1], c='red', label='Positions')
320     plt.title(f'Bandeja {track_id} Trayectoria')
321     plt.xlabel('X position (m)')
322     plt.ylabel('Y position (m)')
323     plt.xlim(x_limits)
324     plt.ylim(y_limits)
325     plt.legend()
326     plt.savefig(f"track_{track_id}_positions.png")
327     plt.close() # Cierra la figura para no consumir memoria
328 
329     # Gráfico de la velocidad de las bandejas
330     plt.figure(figsize=(10, 5))
331     plt.plot(velocities[:, 0], label=f'Tray {track_id} X Velocities')
332     plt.plot(velocities[:, 1], label=f'Tray {track_id} Y Velocities')
333     plt.title(f'Bandeja {track_id} Velocidad')
334     plt.xlabel('Frame')
335     plt.xlim(frame_limits)
336     plt.ylim(velocity_limits)
337     plt.ylabel('Velocity (m/s)')
```

```
338        plt.legend()
339        plt.savefig(f"track_{track_id}_velocities.png")
```

Índice de Figuras

1.1	Puerta automática de Nerón	2
1.2	Prueba de Turing	3
1.3	Modelo teórico del perceptrón	3
1.4	Función de activación sigmoide	4
1.5	Ley de Moore para CPU y memorias	5
1.6	Gemelo digital de un proceso de fabricación	6
1.7	Detección de vehículos en la carretera y su posición respecto a la cámara	8
1.8	Imagen de una manzana con un filtro para detectar irregularidades	9
2.1	Conjunto de piezas	12
2.2	Previsualización de los objetos a imprimir	14
2.3	Imagen del ensamblaje de la Raspberry Pi con su soporte.	14
3.1	Soporte de madera elevado para colocación de la Raspberry Pi	20
3.2	Posición final de la Raspberry Pi 4 con la cámara	21
3.3	Vista general desde altura final con móvil de última generación	21
3.4	Fotograma usado para entrenamiento de red neuronal	22
3.5	Fotograma usado para entrenamiento de red neuronal etiquetada en labelImg	22
3.6	Detección fotograma mediante SSD	24
3.7	Resultados obtenidos del entrenamiento de la red neuronal Yolov5n	25
3.8	Fotograma detección por Yolov5n	26
3.9	Fotograma para seguimiento de las bandejas detectadas	28
3.10	Trackeo bandejas ID 2 vs ID 5	28
3.11	Imágenes para la calibración de la Picamera	30
3.12	Imágenes para la calibración de la Picamera	31
4.1	Fotograma extraído del vídeo final procesado	33
4.2	Posición bandeja en el plano XY real	34
4.3	Velocidad detectada en la bandeja en cada dirección	34

Índice de Tablas

2.1 Comparación entre el Portátil de Alta Gama y la Raspberry Pi 4 16

3.1 Comparación de los tiempos de inferencia media. 24
3.2 Comparación de los tiempos de inferencia medios para diferentes redes neuronales y dispositivos. 26
3.3 Posiciones de referencia para los botones blancos y centros de ArUcos 32

Bibliografía

[1] International Organization for Standardization, “Automation systems and integration - digital twin framework for manufacturing - part 1: Overview and general principles,” ISO 23247-1:2021, 10 2021.

[2] ——, “Automation systems and integration - digital twin framework for manufacturing - part 2: Reference architecture,” ISO 23247-2:2021, 10 2021.

[3] ——, “Automation systems and integration - digital twin framework for manufacturing - part 3: Digital representation of manufacturing elements,” ISO 23247-3:2021, 10 2021.

[4] ——, “Automation systems and integration - digital twin framework for manufacturing - part 4: Information exchange,” ISO 23247-4:2021, 10 2021.

[5] Ultralytics, “Yolov5,” https://github.com/ultralytics/yolov5, 2023.

[6] R. Munawar, “yolov5-object-tracking,” https://github.com/RizwanMunawar/yolov5-object-tracking, 2023.

[7] G. A. Terejanu, “Extended kalman filter tutorial,” http://www.terejanu.com/, 2007.

[8] E. W. Weisstein, “Rodrigues’rotation formula,” http://electroncastle.com/wp/?p=39.

[9] R. Muñoz Salinas, “Aruco: A concise library for augmented reality applications based on fiducial markers,” https://docs.google.com/document/d/1QU9KoBtjSM2kF6ITOjQ76xqL7H0TEtXriJX5kwi9Kgc/edit#heading=h.1z8vm961dhos.

[10] M. R. Arahal, *Apuntes de Sistemas de Percepción*.

[11] *OpenCV Documentation*, https://docs.opencv.org/4.x/, OpenCV, 2023.

[12] OpenCV, “Aruco marker detection,” https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html, 2023.

[13] “Yolov5 benchmark on raspberry pi 4b (arm cortex-a 72),” https://www.researchgate.net/figure/YOLOv5-benchmark-on-Raspberry-Pi-4B-Arm-Cortex-A-72_fig1_362123383, 2023.

[14] E. Twomey, “Using charuco boards in opencv,” https://medium.com/@ed.twomey1/using-charuco-boards-in-opencv-237d8bc9e40d, 2023.

[15] Universidad Complutense de Madrid, “Filtro de kalman,” https://www.ucm.es/data/cont/media/www/pag-41459/Filtro%20de%20Kalman.pdf.

[16] Q Engineering, “Deep learning on raspberry pi and more,” https://qengineering.eu/, 2023.