UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y DISEÑO INDUSTRIAL

Grado en Ingeniería Electrónica Industrial y Automática

TRABAJO FIN DE GRADO

Desarrollo de un sistema de clasificación y colocación de objetos con un brazo colaborativo

Autor: David Tertre Boyé

Tutor: David Álvarez Sánchez

Departamento en Ingeniería Eléctrica, Electrónica Automática y Física Aplicada

Madrid, junio del 2024

AGRADECIMIENTOS

Primero de todo quisiera agradecer a mi familia, en especial a mis padres, hermanos y abuelos, que durante todo el proyecto me han estado apoyando para dar lo mejor de mí.

A todos mis amigos que me han estado ahí siempre y gracias a los cuales he podido llegar donde estoy.

También quisiera agradecer a todos los profesores que me han inspirado y enseñado en estos años de carrera, en especial a David, mi tutor del TFG.

RESUMEN

El presente proyecto de final de carrera realiza un sistema integral de clasificación y colocación de objetos con el brazo robótico UR3e. Se han desarrollado dos programas que actúan simultáneamente para cumplir los objetivos establecidos. El primero se ejecuta en el robot, programa de Universal Robots, y el segundo, programa en Python, en el ordenador con sistema operativo Linux.

El sistema resuelve el problema de varias formas diferentes, haciendo uso de métodos avanzados de procesamiento de imágenes, visión artificial y técnicas de aprendizaje automático para identificar, clasificar y posicionar los objetos presentes en el entorno. Estos métodos incluyen la detección mediante apriltags, el entrenamiento y uso de la red neuronal convolucional YOLOv8, y técnicas clásicas de visión artificial como el filtrado de imágenes y la detección de contornos y esquinas.

Una vez detectados y clasificados, el brazo robótico ejecuta operaciones de *pick and place*, recogiendo los objetos y ubicándolos en sus posiciones designadas.

El sistema ha sido diseñado para funcionar en un entorno controlado, optimizando la iluminación y el contraste entre los objetos y el fondo para mejorar la precisión de los métodos de detección. Además, se ha implementado un manejo robusto de excepciones para garantizar la estabilidad del sistema en caso de desconexiones o mal funcionamiento de los componentes.

La integración del sistema incluye la coordinación entre el robot, la cámara y los algoritmos de detección, permitiendo una operación fluida y eficiente.

Palabras clave:

Robot, UR3e, Python, técnicas, métodos, visión artificial, red neuronal convolucional, YOLO, detección, sistema, coordinación, pick and place.

ABSTRACT

The present final project develops a comprehensive system for object classification and placement using the UR3e robotic arm. Two programs have been developed to simultaneously fulfil the established objectives. The first runs on the robot, a Universal Robots program, while the second, a Python program, runs on a Linux operating system computer.

The system addresses the problem in various ways, employing advanced methods of image processing, computer vision, and machine learning techniques to identify, classify, and position objects in the environment. These methods include detection using apriltags, training and use of the YOLOv8 convolutional neural network, and classical computer vision techniques such as image filtering and contour and corner detection.

Once detected and classified, the robotic arm performs pick-and-place operations, picking up the objects and placing them in their designated positions.

The system has been designed to operate in a controlled environment, optimizing lighting and contrast between objects and the background to improve the accuracy of detection methods. Additionally, robust exception handling has been implemented to ensure system stability in case of disconnections or malfunctions of the components.

System integration involves coordination between the robot, the camera, and the detection algorithms, enabling smooth and efficient operation.

Keywords:

Robot, UR3e, Python, techniques, methods, computer vision, convolutional neural network, YOLO, detection, system, coordination, pick and place.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

LISTA DE ACRÓNIMOS

Acrónimo	Significado
API	Interfaz de Programación de aplicaciones
GUI	Interfaz Gráfica de Usuario
RTDE	Intercambio de Datos a Tiempo Real
TCP	Punto Central de la Herramienta
IMU	Unidad de Medición Inercial
TOPS	Tera Operaciones Por Segundo
FF	Enfoque Fijo
FA	Enfoque automático
RNA	Red Neuronal artificial
CNN	Red Neuronal Convolucional
YOLO	Solo mira una vez (*You Only Look Once*)
RELU	Unidad Lineal Rectificada
MLP	Perceptrón multicapa
RNN	Red neuronal recurrente
COCO	Objetos Comunes en Contexto
VOC	Clases de Objetos Visuales
UML	Lenguaje unificado de modelado
GFLOPS	Operaciones de Punto Flotante por Segundo

Capítulo 1. INTRODUCCIÓN

1.1. MOTIVACIÓN DEL PROYECTO

El motivo principal del proyecto es personal, para ampliar los conocimientos adquiridos durante la formación académica, profundizando en tecnologías y áreas de investigación que experimentan un auge significativo en la actualidad. Dos de las ramas más destacadas son los robots colaborativos, en el campo de la robótica, la visión artificial, y el uso de la inteligencia artificial en esta disciplina.

La finalidad es abordar problemas de la actualidad, uniendo diferentes campos y aplicando lo aprendido de manera creativa e innovadora. Al integrar los principios de los robots colaborativos con las capacidades de la visión artificial, se pueden diseñar sistemas inteligentes y adaptables que sean capaces de interactuar segura y eficientemente en entornos y aplicaciones.

Esta inmersión en áreas de vanguardia no solo ampliará mis habilidades técnicas y mi comprensión teórica, sino que también me permitirá crecer como ingeniero al enfrentarme a desafíos complejos y a contribuir al avance y desarrollo de tecnologías innovadoras que impactan positivamente en la sociedad.

1.2. ESTADO DEL ARTE

1.2.1. ROBOTS COLABORATIVOS

Los robots colaborativos, también llamados cobots, están diseñados para operar junto con humanos de manera segura y eficiente, asistiendo en diversas tareas y procesos. A diferencia de los robots industriales, los cobots se construyen específicamente para interactuar en espacios compartidos, mejorando las capacidades humanas de múltiples formas.

Orígenes

El término "cobot" fue acuñado por Edward Colgate y Michael Peshkin, profesores de la Universidad Northwestern. Fueron los primeros inventar un cobot, mostrando su diferenciador principal de los robots ya existentes en el mundo: un "cobot" es un dispositivo robótico que manipula objetos en colaboración con un operador humano [1]. El robot colaborativo brinda asistencia al operario configurando superficies virtuales que pueden usarse para restringir y guiar el movimiento.

Les tomó pocos años a otras empresas para lanzar sus propios modelos de cobots, como Universal Robots, actualmente el proveedor más grande de cobots en Europa.

Beneficios

Mejora de las capacidades humanas: Los cobots están diseñados para realizar trabajos manuales, repetitivos o de riesgo, con mayor precisión y fuerza que los humanos. Son muy útiles ya que reducen el número de lesiones y accidentes graves a los operarios y les alivian de actividades monótonas.

Características distintivas: Presentan las siguientes ventajas que los diferencian de los robots industriales convencionales:

Inversión: Son más económicos y, a largo plazo, pueden llegar a ser más rentables por su gran flexibilidad y adaptación a los cambios empresariales.

Fácil instalación y programación: Pueden estar operativos en poco tiempo debido a la utilización de aplicaciones móviles y software de escritorio. Además, pueden ser programados para diferentes tareas, mientras que el robot no colaborativo está diseñado con un único propósito.

Compacto: El tamaño reducido permite su uso en cualquier proceso de producción.

Móvil: Al ser más compactos y pequeños, su transporte es más sencillo, a diferencia de los robots industriales.

Flexible: Su facilidad de programación permite configurar fácilmente nuevas operaciones, lo que permite su utilización en diferentes etapas de producción.

Seguridad: Están diseñados para cooperar de manera segura en cualquier proceso. Los sensores integrados realizan las tareas necesarias para garantizar la protección del personal sin la necesidad de barreras adicionales.

Gracias a estas características, Los cobots son una buena herramienta de trabajo. Destaca su amplia utilización por las pequeñas y medianas empresas (PYMEs), debido a su facilidad de uso, el coste accesible y la adaptabilidad, haciéndoles ideales para diversas aplicaciones industriales [2].

Conclusión

Los robots colaborativos han sido desarrollados para facilitar el trabajo, permitiendo a los operarios enfocar la atención en tareas más organizativas y estratégicas. En lugar de reemplazar a los humanos con partes autónomas, los robots colaborativos mejoran las capacidades como precisión, procesamiento de información o capacidad física.

Este enfoque es esencial para la industria 4.0, que busca integrar la tecnología avanzada en la producción para crear entornos más eficientes y seguros, aportando un valor significativo a las organizaciones.

1.2.2. VISIÓN ARTIFICIAL

La visión artificial es un campo encargado de intentar replicar la percepción visual humana, para que los sistemas tengan la capacidad de adquirir, procesar, analizar y comprender imágenes y videos digitales.

Orígenes

La visión artificial surgió en la década de1960 [3]. Los investigadores empezaron a explorar la idea de que los ordenadores pudieran interpretar imágenes. Los primeros esfuerzos se basaron en la digitalización de imágenes y el desarrollo de algoritmos para tareas básicas como la detección de bordes. La falta de poder computacional y los limitados datos (pocas imágenes) restringió significativamente estos avances.

En la década de los 70 se introdujeron algoritmos fundamentales como la transformada de hought para la detección de líneas y la trasformada de fourier para el análisis de frecuencias en imágenes. También se desarrollaron métodos para el análisis del movimiento y la reconstrucción en 3d con múltiples imágenes.

En la década de los 2010 comenzó la verdadera revolución gracias a los avances en el aprendizaje profundo. La aparición de las redes neuronales convolucionales (RCCs) transformó el campo permitiendo alcanzar niveles de precisión sin precedentes.

Inteligencia artificial

La inteligencia artificial (IA) es un campo de la informática que se enfoca en crear sistemas que puedan realizar tareas que normalmente requieren inteligencia humana, como el aprendizaje, el razonamiento y la percepción.

Estas técnicas incluyen el reconocimiento de voz, la toma de decisiones, la resolución de problemas y la compresión del lenguaje natural.

Se puede subdividir en dos categorías, IA débil, algoritmos diseñados para realizar una tarea específica, y IA fuerte, haciendo referencia a sistemas con capacidades cognitivas generales, comparables con las del ser humano.

Aprendizaje automático

El aprendizaje automático es un subcampo de la IA que se centra en el desarrollo de algoritmos que permiten a un sistema su aprendizaje y mejora a partir de la experiencia. En lugar de ser programadas para una única tarea, estos sistemas utilizan sus datos para construir modelos que producen o toman decisiones sin intervención humana directa. Existen tres tipos principales de aprendizaje automático:

Aprendizaje supervisado: El modelo es entrenado con datos etiquetados, por lo que se conocen las entradas y salidas deseadas.

Aprendizaje no supervisado: El modelo se entrena con datos sin etiquetar, así que se encarga de buscar patrones o estructuras que pueda relacionar.

Aprendizaje por refuerzo: El modelo se entrena con datos sin etiquetar, aprende con la experiencia. Recibe señales de premio o castigo en función de su respuesta, lo que le lleva a ajustar su comportamiento para maximizar las recompensas.

Aprendizaje profundo

El aprendizaje profundo (*deep learning*) es una subcategoría del aprendizaje automático que se caracteriza por la utilización de redes neuronales con muchas capas. Estas redes, al tener capas, aprenden representaciones de datos en múltiples niveles de abstracción, lo que las hace especialmente efectivas en tareas complejas como el reconocimiento de imágenes y el procesamiento del lenguaje natural.

Visión artificial

A lo largo de estos últimos años, la evolución de la visión artificial ha sido imparable. Esto se debe, por una parte, a la mejora del hardware, cada vez más rápido, preciso y con mayor resolución. Por otra parte, a la mejora del software y los algoritmos de análisis de imágenes.

El desarrollo de algoritmos y técnicas en visión artificial ha experimentado avances exponenciales las últimas décadas, impulsados por el crecimiento de la potencia computacional, la disponibilidad de grandes conjuntos de datos y los avances en redes neuronales profundas. Estos avances han permitido abordar muchos problemas que hasta ahora tenían una difícil solución.

La intersección entre la visión artificial y el aprendizaje profundo ha sido en núcleo de estos numerosos avances. En particular, la adopción generalizada de las redes neuronales convolucionales ha transformado este campo, permitiendo un rendimiento sin precedentes en términos de precisión y velocidad.

En el ámbito especifico de la detección de objetos, las CNNs han demostrado ser especialmente efectivas. Modelos como R-CNN y YOLO han establecido estándares en términos de precisión y velocidad, mejorando la eficiencia, permitiendo el análisis a tiempo real.

Se ha transformado la capacidad de obtener y procesar información visual de manera automatizada y precisa.

1.3. OBJETIVOS

- Implementar un sistema de comunicación y control de movimiento del brazo robótico de Universal Robots desde Python y Linux.

- Implementar un sistema de visión artificial que permita detectar, situar en el espacio tridimensional y clasificar una serie de objetos conocidos con distinta geometría.
- Implementar un método que coja los objetos que se encuentran en una determinada zona del espacio de trabajo y los coloque en otro punto del espacio de trabajo preestablecido.

1.4. ESTRUCTURA DEL DOCUMENTO

Para facilitar la lectura, se detalla el contenido de cada capítulo.

- El Capítulo 1 es el presente capítulo, contiene la introducción, el estado del arte, los objetivos y la estructura del proyecto.
- El Capítulo 2 explica los fundamentos teóricos necesarios para comprender el funcionamiento del proyecto.
- El Capítulo 3 muestra el desarrollo del proyecto, formado por el entorno de trabajo y desarrollo, así como la implementación de la cámara, el robot y los modelos de visión artificial para alcanzar el objetivo propuesto.
- El Capítulo 4 analiza los resultados, comparando las diferentes formas seguidas a la hora de obtener el resultado final.
- El Capítulo 5 es el último capítulo, contiene las conclusiones del proyecto, las áreas a mejorar y posibles extensiones para la realización de futuros proyectos.

Capítulo 2. FUNDAMENTOS TEÓRICOS

2.1. UR3E

El UR3e, de la serie e-Series de Universal Robots, es un robot colaborativo de sobremesa pequeño, perfecto para tareas de ensamblaje ligeras y bancos de trabajo automatizados. El formato compacto permite su uso en espacios reducidos. Puede ser colocado en mesas de trabajo o instalarse en maquinaria, mostrando su flexibilidad y capacidad de trabajar con humanos.

Figura 2.1 Robot UR3e [4]

2.1.1. Estructura y Especificaciones del robot

El Ur3e es un robot articulado de seis grados de libertad. Todas las articulaciones son giratorias, con un rango de trabajo de +-360º, excepto la última, que presenta un giro infinito, apto para tareas como el atornillado.

La Base es robusta y proporciona estabilidad al brazo. Contiene el primer eje de rotación permite el giro alrededor suyo. El segundo eje es el hombro, el cual permite el movimiento hacia delante y hacia atrás. El Codo es el tercer eje, permitiendo el movimiento de extensión y retracción. Finalmente, le siguen tres muñecas encargadas de los movimientos más precisos. En la última muñeca se instala la herramienta.

A su vez, el robot lleva integrado una caja de control y una consola de programación.

Cuenta con diecisiete funciones de seguridad configurables, y certificaciones en ISO 12849-1, PLd (*Performance Levels*) Categoría 3, y en ISO 10218-1. A continuación se muestran dos tablas que detallan las especificaciones técnicas y las características físicas del cobot.

Tabla 2.1 Especificaciones UR3e [13]

Especificación	**Valor**
Carga útil	3 kg
Alcance	500 mm
Grados de libertad	6 articulaciones giratorias
Programación	Pantalla táctil 12'' con GUI PolyScope

Tabla 2.2 Características Físicas UR3e [13]

Característica	**Valor**
Huella	Ø 128 mm
Peso con cable	12,2 kg
Materiales	Aluminio, plástico, acero
Rango de temperatura de funcionamiento	0-50 ºC
Tipo de conector para herramientas	M 8 | M8 8-pin

Estas características únicas convierten al UR3 en uno de los robots colaborativos de sobremesa más flexibles y ligeros del mercado, apto para trabajar codo con codo con los empleados.

2.1.2. GRIPPER HRC-03

El HRC-03-118505 de Zimmer Group es una pinza colaborativa. Está diseñada específicamente para para su uso en actividades como la manipulación de piezas. Es compatible con varios modelos de robots aparte del ur3e, facilitando su integración en diferentes sistemas.

Figura 2.2 Gripper HRC-03 [5]

La gestión de cables es interna, para una mayor organización y seguridad. El tipo de acondicionamiento es eléctrico y el control es I/O (entrada/salida). Las características físicas más importantes están en la siguiente tabla.

Tabla 2.3 Características HCR-03 [5]

Característica	**Valor**
Peso	0.68 kg
Voltaje	24 V
Rango de temperatura de funcionamiento	5-50 ºC
Tiempo de apertura o cierre	0.19 s
Carrera por mandíbula	10 mm
Fuerza de agarre máxima ajustable	190 N

2.1.3. PolyScope

PolyScope es una interfaz gráfica de usuario (GUI) utilizada por los robots colaborativos de Universal Robots. Esta interfaz permite manejar el brazo robótico y la caja de control, así como crear programas y ejecutarlos fácilmente sin la necesidad de conocimientos avanzados en programación.

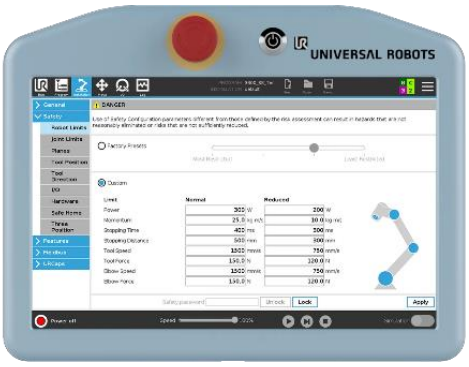

Figura 2.3 PolyScope 5 [4]

2.1.4. Real Time Data Exchange (RTDE)

RTDE es una interfaz de intercambio de datos en tiempo real desarrollada por Universal Robots [14][4]. Proporciona una forma de sincronizar aplicaciones externas con el controlador UR sobre una conexión TCP/IP estándar, sin romper funciones en tiempo real del controlador. La interfaz RTDE está disponible por defecto cuando el controlador está funcionando.

La sincronización es ajustable, lo que significa que podemos configurar la entrada y la salida de datos. Estas variables deben estar contenidas en los paquetes de sincronización de datos reales.

La funcionalidad se puede dividir en dos etapas: un procedimiento de configuración y un bucle de sincronización.

2.1.5. Registros

Los registros son variables internas de almacenamiento que permiten guardar y manipular diferentes tipos de datos durante la programación y operación del robot. Estos registros son esenciales para la flexibilidad y funcionalidad del robot.

Cuando se inicia el bucle de sincronización, la interfaz RTDE envía al cliente los datos solicitados en el mismo orden que el cliente solicitó. Además, se espera que el

cliente envíe entradas actualizadas a la interfaz RTDE sobre un cambio de valor. La sincronización de datos utiliza datos binarios en serie. Tanto el cliente como el servidor pueden en cualquier momento enviar un mensaje de texto, en el que el nivel de advertencia especifica la gravedad del problema. El IDE está disponible en el puerto número 30004.

2.1.6. Biblioteca Python RTDE

Universal Robots ha creado una biblioteca para proporcionar una API que permite interactuar con la interfaz de intercambio de datos en tiempo real (RTDE). Esta biblioteca facilita la programación y control de los robots de Universal Robots, permitiendo el envío y recepción de datos en tiempo real entre el robot y una aplicación externa.

La biblioteca ha sido desarrollada en Python, pero no está disponible en el repositorio oficial de PyPI (Python Package Index). Esto implica que no se puede instalar en el entorno virtual del proyecto a través de administrador de paquetes de Python (pip), sino que es necesario clonar el repositorio de GitHub para su instalación.

2.2. OAK-D-Lite

La OAK-D-LITE es una cámara fabricada por Luxonis [6]. La cámara cuenta con 3 cámaras integradas. Una cámara central RGB (13MP), de alta resolución, apta para capturar imágenes y videos nítidos. Dos cámaras de profundidad en los lados(480P), monoStereoDepth, destinadas para obtener información precisa de la profundidad y distancia de los objetos del entorno.

Figura 2.4 Cámara Luxonis OAK-D-Lite [6]

2.2.1. ESPECIFICACIONES TÉCNICAS

La cámara OAK-D Lite es más pequeña, ligera y utiliza menos energía que la OAK-D. Se conecta a través de un cable USB-C utilizando el rango máximo. La alimentación se suministra por el mismo cable. Esto hace que la conexión sea sencilla y conveniente. A continuación, se muestra una tabla con las características físicas.

Tabla 2.4 Características Físicas [16][6]

Característica	**Valor**
Peso	0.061 kg
Anchura	91 mm
Altura	28 mm
Profundidad	17.5 mm

La cámara RGB puede ser de enfoque fijo (FF) o enfoque automático (AF), dependiendo de la versión comprada.

El rango de temperaturas de funcionamiento va desde los -30ºC hasta los 70ºC, teniendo en cuenta que la temperatura varía entre módulos.

Tiene integrado un controlador BMI270, una Unidad de Control Inercial (IMU) de seis ejes (x, y, x). Tres de velocidad angular y tres de aceleración.

Cuenta con un módulo de visión artificial RVC2 (*Robotics Vision Core 2*), cuya potencia de porcesamiento es de 4 TOPS (Tera Operaciones por segundo). Tiene una serie de funciones clave como son:

- Ejecución de modelos IA personalizados, aunque es necesario convertir los modelos a un formato '*MyriadX .blob'*.
- Visión por computadora, pudiendo manipular imágenes, identificar bordes y contornos, y seguimiento de características personalizadas.
- Percepción de profundidad estéreo y alineación RGB-profundidad, permitiendo la generación de una nube de puntos con el nodo '*PointCloud*'.
- Seguimiento de objetos con el nodo '*ObjectTracker*'.

2.2.2. Bibioteca Python DepthAI

La biblioteca Python DepthAI, desarrollada por Luxonis, proporciona una interfaz de enlace con la biblioteca depthai-core escrita en C++ [15]. Es una herramienta poderosa que permite interactuar con la cámara OAK-D Lite de forma programática dentro del entorno de desarrollo Python.

La biblioteca proporciona una interfaz que facilita el acceso a los módulos de la cámara y a las capacidades avanzadas, como la detección de objetos en tiempo real, el seguimiento de objetos y la generación de nubes de puntos con unos nodos predefinidos.

2.3. Apriltags

Apriltags es un sistema de etiquetas visuales desarrollado por investigadores de la Universidad de Michigan para proporcionar una localización de alta precisión y baja sobrecarga de procesamiento [7].

Es útil en una amplia variedad de tareas que incluyen realidad aumentada, robótica y calibración de cámaras. El objetivo principal del software de detección AprilTag es calcular la identidad, posición y orientación 3D precisas de las etiquetas en relación con la cámara [19]. La biblioteca AprilTag está implementada en C sin dependencias externas. El diseño de la biblioteca ha sido pensado para poder incluirse fácilmente en otras aplicaciones o dispositivos integrados. Gracias a la poca carga computacional se puede lograr rendimiento en tiempo real incluso en procesadores de baja calidad.

Existen otros tipos de marcadores, como ARToolkit, DataMatrix y códigos QR. Cada uno presentan distintas características en el patrón fiduciario, sistema de codificación y robustez, corrección de errores, capacidad de información y aplicaciones principales.

2.3.1. Descripción técnica

Los apriltags son un tipo de código de barras de dos dimensiones. Presentan un patrón fiduciario simple y de alto contraste. La estructura del marcador es un cuadrado dividido en una matriz de celdas más pequeñas. La estructura típica depende del tipo específico de apriltag.

El borde de localización se encuentra alrededor de la matriz central y este ayuda al algoritmo de detección a identificar los límites del marcador y determinar su orientación.

Los apriltags cuentas con seis familias. Cada una con unas características diferentes, como la dimensión de la cuadricula, que determina la capacidad de almacenamiento. A continuación, se muestran las diferentes familias con sus respectivos nombres.

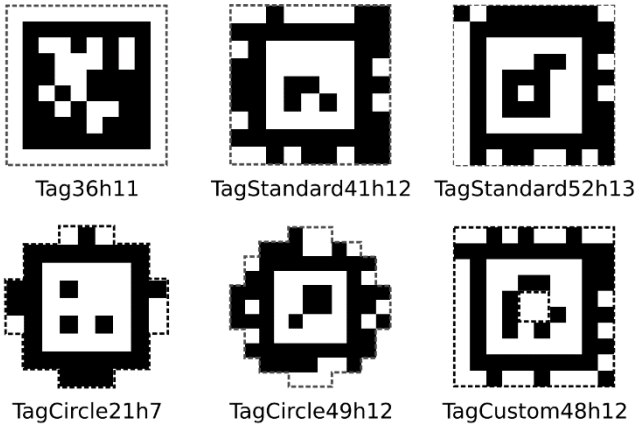

Figura 2.5 Familias de apriltags [7]

El tipo de codificación es binaria, las celdas de dentro del bloque de localización se utilizan para codificar la información en forma de bits. Cada celda puede ser negra (0) o blanca (1), creando un patrón único en cada marcador.

Cada patrón codifica un único número, que se utiliza como identificador del marcador, causando la unicidad y la distinción entre otros marcadores.

El sistema de codificación lexicográfica, método de organización de los códigos, busca la mayor diferencia entre los códigos para minimizar la posibilidad de confusión entre dos marcadores, a pesar de que se encuentren parcialmente dañados o la imagen tenga ruido. Esta distancia entre códigos válidos, llamada distancia de *Hamming*, es maximizada para reducir la probabilidad de error en la lectura y así mejorar la capacidad de corrección de errores.

El sistema de codificación incluye la redundancia para así poder corregir errores. El algoritmo puede reconstruir el patrón correcto basándose en la información redundante.

A parte, los algoritmos están diseñados para ser robustos frente a distorsiones, inclinaciones y variaciones de iluminación. Utilizan técnicas avanzadas de procesamiento de imágenes para localizar el marcador, ajustar la perspectiva y decodificar el patrón binario de manera precisa.

2.3.2. BIBLIOTECA PYTHON PUPIL-APRILTAGS

Pupil-apriltags es una biblioteca escrita en Python que proporciona enlaces a la biblioteca Apriltags versión 3 desarrollada en C++ por por AprilRobotics. Es una versión mejorada que soluciona el error de compilación de ruedas (*wheels*) en la descarga de la librería mediante el pip en Windows.

Proporciona una interfaz que facilita la detección de apriltags en imágenes y proporciona datos de interés como la pose respecto de a la cámara.

2.4. REDES NEURONALES CONVOLUCIONALES

2.4.1. PRINCIPIO Y FUNCIONAMIENTO DE LAS REDES NEURONALES

Las redes neuronales artificiales (RNA) se definen como sistemas de mapeos no lineales cuya estructura se basa en principios observados en los sistemas nerviosos de humanos y animales [8]. Están formadas por un número grande de procesadores simples ligados por conexiones con pesos.

Neurona artificial

La neurona artificial es la unidad de procesamiento. Cada una se compone de varias partes.

- Entradas (x_1, x_2, …, x_n): valores que recibe la neurona de otros nodos.
- Pesos (w_1, w_2, …, w_n): valores por los que se multiplica la entrada, determinando su importancia. Si el peso es positivo se habla de una excitación y si es negativo se considera una inhibición.
- Suma ponderada ($\boldsymbol{u}$): Se suma las entradas por los pesos, más la adición de un sesgo/bias ($\boldsymbol{\theta}$).

$$\boldsymbol{u} = \sum_{i=1}^{n} \boldsymbol{w_i x_i} + \boldsymbol{\theta}$$

Ecuación 2.1 Suma ponderada

- Función de activación (F): La suma ponderada pasa por una función de activación, que introduce la no linealidad y decide la salida final de la neurona (y),

$$y = F(u)$$

Ecuación 2.2 Ecuación de salida

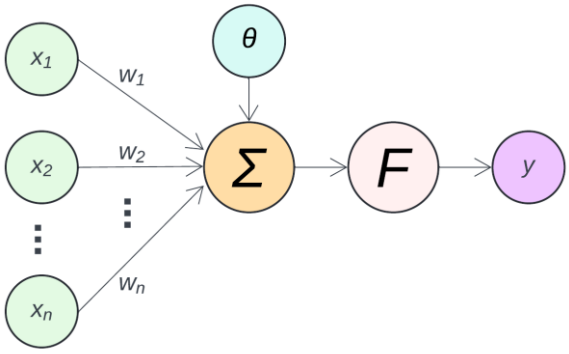

Figura 2.6 Modelo de neurona artificial estándar

Este modelo matemático es aplicado en todas las neuronas de la red para calcular su salida, que puede ser entrada para las neuronas de la siguiente capa. Las funciones de activación más comunes incluyen la función escalón, sigmoide, tangente hiperbólica, de gauss y ReLU (*Rectified Linear Unit*).

El perceptrón, creado por Frank Rosenblatt en 1958, es uno de los primeros modelos de neurona artificial [8]. Los perceptrones de una capa pueden clasificar correctamente conjuntos de datos que son linealmente separables, esto quiere decir que pueden ser separados por un hiperplano. Siguen el modelo matemático descrito anteriormente, utilizando para su activación una función de no linealidad binaria (escalón).

Red neuronal

Una red neuronal es un conjunto de neuronas artificiales agrupadas en capas. Las principales capas son:

- Capa de entrada: Donde se introducen los datos.
- Capas ocultas: Una o más capas entre la entrada y la salida. Donde se realizan la mayoría de los cálculos y transformaciones.
- Capa de salida: el resultado final.

Cada capa está completamente conectada con la siguiente. Lo que quiere decir que la salida de cada neurona se conecta con todas las neuronas de la siguiente capa.

Entrenamiento de la red neuronal

Se denomina entrenamiento al proceso de configuración de una red neuronal, donde se ajustan los pesos y sesgos para que las entradas produzcan salidas deseadas. Un modo de lograrlo es estableciendo los pesos conocidos con anterioridad. El otro método implica técnicas de retroalimentación y patrones de aprendizaje, es decir, un algoritmo de optimización, como puede ser el descenso de gradiente.

Durante el entrenamiento la red minimiza una función de pérdida que mide el error entre las predicciones de la red y los valores reales. Además, el aprendizaje puede dividirse en supervisado o asociativo, no supervisado o auto-organizado y por refuerzo, explicados más detalladamente en el apartado 1.2.2.

Tipos de redes

Existen diferentes arquitecturas según el propósito general de la red. Las más comunes son:

Perceptrón multicapa (MLP): Compuesto por capas totalmente conectadas entre ellas. Se suele utilizar en problemas de regresión y clasificación.

Redes recurrentes (RNN): Adecuadas para datos secuenciales, como series temporales o el procesamiento del lenguaje natural.

Redes convolucionales (CNN): Específicamente diseñadas para procesar datos con estructura de rejilla, como imágenes y videos.

2.4.2. REDES NEURONALES CONVOLUCIONALES

Las redes neuronales convolucionales (CNN) son un tipo de red neuronal artificial. Están inspiradas en la organización y funcionamiento del sistema visual biológico [20].

Este tipo de red es una variación avanzada del perceptrón multicapa, diseñada específicamente para para el procesamiento de datos estructurados en matrices bidimensionales (ej. imagen).

Esta adaptación las hace excepcionalmente eficaces en tareas de visión artificial, como la clasificación y segmentación de imágenes, entre otras aplicaciones. Al utilizar operaciones de convolución, las CNN pueden identificar y aprender características espaciales locales en las imágenes, lo que las convierte en una herramienta poderosa para analizar y comprender el contenido visual de manera automatizada.

Arquitectura

La CNN consta de una capa de entrada, una capa de salida y varias capas ocultas entre ambas. Estas capas realizan operaciones que modifican los datos con el propósito de comprender sus características particulares. Las tres capas más comunes son:

- Capas convolucionales: aplican un conjunto de filtros convolucionales a las imágenes de entrada, cada filtro activa diferentes características de una imagen.
- Capas de activación: aplican una función no lineal, la más común es la Unidad lineal rectificada (ReLU), encargada de mantener los valores positivos y establecer los valores negativos a cero. Solo las características activadas prosiguen a la siguiente capa, permitiendo un entrenamiento más rápido y eficaz.
- Capas de agrupamiento (*Pooling*): Simplifican las salidas mediante reducción no lineal de la tasa de muestreo, lo que disminuye el número de parámetros que la red debe aprender. Reducen la dimensionalidad y aumenta la invarianza a pequeñas transformaciones.

Estas operaciones se repiten en decenas de capas. Cada etapa aprende a identificar diferentes características.

A diferencia de una red neuronal tradicional, una CNN tiene pesos y valores de sesgos compartidos, que son los mismos para todas las neuronas ocultas de una capa determinada. Esto significa que todas las neuronas ocultas detectan las mismas características, tales como bordes o formas, en diferentes regiones, haciendo la red tolerante al tamaño y posicionamiento de los objetos dentro de una imagen.

2.4.3. YOLO (YOU ONLY LOOK ONCE)

YOLO es modelo de red neuronal convolucional de código abierto, diseñado específicamente para la detección precisa de objetos en imágenes. Este algoritmo utiliza una única red neuronal convolucional para realizar la detección en una sola pasada.

Esta arquitectura permite una detección rápida y eficiente de objetos, siendo especialmente útil en aplicaciones que requieren un procesamiento a tiempo real.

Al aprender representaciones generalizables de objetos durante el entrenamiento, permiten un bajo error de detección incluso para entradas nuevas diferentes al conjunto de entrenamiento.

2.4.4. YOLOv8. ULTRALYTICS

Ha sido la red elegida para el desarrollo del proyecto. Es la versión más reciente de YOLO, desarrollada por la empresa Ultralytics. Tiene una amplia documentación apoyada con videos explicativos [9]. Ultralytics provee una librería específica en Python que ha sido utilizada en el proyecto. Facilita tanto el entrenamiento de la red, como su posterior validación, exportación y detección. Se pueden adquirir redes preentrenadas para facilitar en entrenamiento, explicado posteriormente.

Aparte, hay variantes por tamaño (nano, pequeña, mediana, grande y extragrande) y por tipo de análisis (clasificación, detección, pose (puntos clave), segmentación, seguimiento y detección orientada de objetos). Las variantes entrenadas y usadas has sido detección de objetos y estimación de pose. Tiene mejoras respecto a las versiones anteriores en cuanto a precisión y velocidad en todos los tamaños, como se observa en la siguiente imagen.

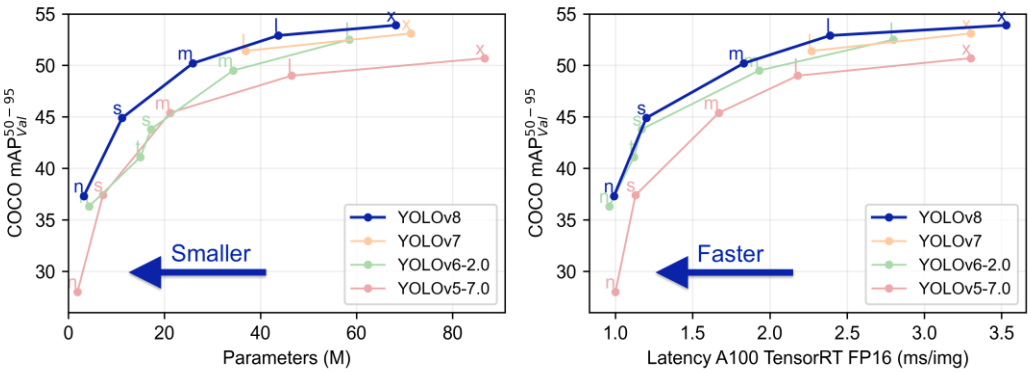

Figura 2.7 Comparación de versiones YOLO [9]

Modelo de Detección de Objetos:

Es el modelo más sencillo para detectar objetos. Los resultados de la detección incluyen:

- Clasificación de objetos: Identificación de la clase de cada objeto.
- Ubicación en la imagen: Coordenadas del recuadro delimitador del objeto (*bounding box*).
- Confianza de detección: Nivel de certeza del modelo sobre la presencia y la clase del objeto.

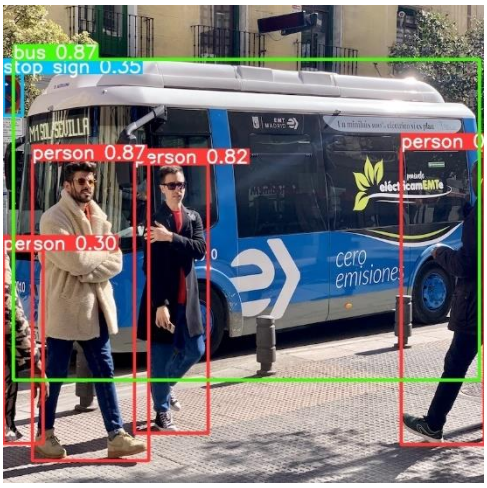

Figura 2.8 Detección YOLOv8

Modelo de Estimación de Pose

Además de proporcionar los datos del modelo de detección de objetos, este modelo incluye la posición de una serie de puntos clave para cada objeto, con los que se ha entrenado previamente el modelo. La identificación de estos puntos clave puede revelar información valiosa sobre la imagen, como la posición, la orientación, y la distancia relativa del objeto.

- Keypoints: Contiene los puntos clave detectados para cada objeto

Figura 2.9 Estimación de Pose YOLO

Entrenamiento

Ultralytics provee en sus recursos ofrecidos en GitHub una serie de cuadernos de Google Colab adaptados para el entrenamiento de la red.

En el proyecto se aplica el concepto de aprendizaje por transferencia (*transfer learning*) para mejorar la eficiencia y eficacia del entrenamiento del modelo. Es una

técnica en la que un modelo previamente entrenado en una tarea se reutiliza y adapta para una nueva tarea. Esto conlleva una serie de beneficios que son:

- Reducción de datos necesarios: el modelo ha aprendido previamente características fundamentales de una amplia variedad de objetos.
- Menor tiempo de entrenamiento: Al utilizar un modelo preentrenado, se necesitan menos épocas de entrenamiento, lo que reduce significativamente el tiempo total.
- Mejora de resultados: al comenzar con una red con conocimientos previos de numerosas clases de objetos, el modelo puede lograr una mejor precisión y desempeño con menos esfuerzo de ajuste.

Para el entrenamiento del modelo personalizado, YOLOv8 utiliza un modelo preentrenado basado en conjuntos de datos como COCO (*Common Object in Context*), VOC (*Pascal visual Object Classes*) e ImageNet. Estos conjuntos de datos son reconocidos por su diversidad y amplitud en imágenes etiquetadas para tareas de detección de objetos, clasificación y segmentación. Estos pesos preentrenados se descargan automáticamente la primera vez que se usa el modelo, facilitando el proceso de configuración y asegurando que el entrenamiento personalizado se beneficie de una base robusta ya establecida.

El uso de esta técnica ha permitido optimizar los recursos disponibles y mejorar los resultados del sistema de detección y posicionamiento de objetos desarrollado en este proyecto. Esta técnica ha demostrado ser esencial para alcanzar una alta eficiencia y efectividad en el entrenamiento del modelo, logrando un balance óptimo entre precisión y tiempo de entrenamiento.

YOLOv8 posee características que lo hacen especialmente bueno en el entrenamiento de la red en diferentes aspectos. Es capaz de utilizar múltiples GPUs para agilizar el proceso. Es fácil de usar, con interfaces sencillas en CLI y Python.

Existe una amplia personalización de hiperparámetros para afinar el rendimiento, la velocidad y la precisión del proceso. Además, de la elección del optimizador, la función de perdida y la composición del conjunto de datos de entrenamiento que influyen significativamente. Tiene seguimiento a tiempo real de las métricas de formación y visualización del proceso de aprendizaje para una mejor comprensión.

Es capaz de reanudar procesos interrumpidos, cargando los pesos del modelo guardado y restaurando el estado del optimizador, el programador de la tasa de aprendizaje y el número de épocas.

Resultados de entrenamiento

En el entrenamiento se obtienen una serie de resultados y métricas que permiten evaluar el rendimiento del modelo. A continuación, se explican las más características:

- *box_loss*: Pérdida asociada con la predicción de las coordenadas de las cajas delimitadoras.
- *pose_loss*: Pérdida asociada con la predicción de la pose o postura de los objetos.
- *kobj_loss*: Pérdida asociada con la detección de objetos en la imagen.
- *cls_loss*: Pérdida asociada con la clasificación de los objetos detectados en las clases correctas.
- *dfl_loss*: Pérdida asociada con la detección de puntos clave o *landmarks* en objetos.

Capítulo 3. DESARROLLO DEL PROYECTO

3.1. ENTORNO DE LABORATORIO

El núcleo del entorno de laboratorio es el brazo robótico, situado en el centro de la mesa, sin obstáculos y con un amplio espacio para maniobrar. Se ha creado un soporte diseñado específicamente para la cámara. Está fijado en el último eslabón del robot con una abrazadera. El soporte proporciona una plataforma estable para fijar la cámara con dos tornillos que van por la parte trasera. Una vez la cámara está fija, queda orientada con las lentes en dirección paralela al eje Z de la herramienta del robot, como se puede observar en la siguiente imagen.

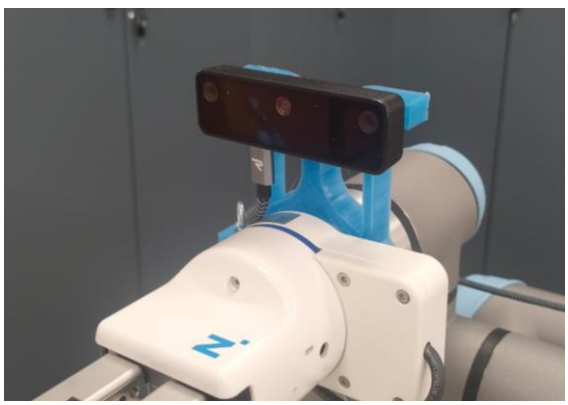

Figura 3.1 Efector final del robot con la cámara acoplada

En la plataforma de soporte del robot, hay un área destinada al almacenamiento de piezas, donde el robot las recoge para su manipulación. En otra zona, al alcance del robot, se encuentra un tablero con huecos que corresponden a las formas de las piezas. Cada pieza tiene tres huecos, con dimensiones ligeramente diferentes: el primero con un 5% de aumento, el segundo con un 10%, y el último con un 15%. Estas variaciones permiten medir y analizar el error de arrastre cometido en cada etapa del proceso, desde la detección hasta la colocación de la pieza, asegurando así que el sistema puede manejar diferentes niveles de tolerancia y garantizar una colocación precisa de las piezas.

Por último, contamos con la Tablet del robot y un ordenador con sistema operativo Linux, conectado a la red local del laboratorio mediante un cable ethernet. El robot debe estar conectado a la red para la correcta conexión entre ambos, mediante el protocolo TCP/IP.

Figura 3.2 Entorno de laboratorio

3.2. ENTORNO DE DESARROLLO

Como se ha dicho anteriormente, el laboratorio tiene un ordenador con sistema operativo Linux, del que se ha desarrollado gran parte del código del proyecto. Además, el proyecto es compatible en sistemas operativos Windows.

Para la edición de texto, se ha optado por utilizar Visual Studio Code (VS Code), complementado con su extensión para Python, que facilita la visualización de errores y la ejecución el programa.

Para la gestión de bibliotecas y la correcta ejecución del programa se lleva a cabo a través del entorno virtual de Python, que se encarga de mantener todas las dependencias del proyecto y de aislarlas del entorno Python del sistema.

Se ha utilizado GitHub para el control de versiones del proyecto. Se ha creado un repositorio que permite el desarrollo colaborativo en varios ordenadores. Se puede acceder al repositorio del proyecto a través del enlace puesto en el Anexo B.1.

3.3. IMPLEMENTACIÓN DEL BRAZO ROBÓTICO

El brazo robótico colaborativo utilizado en el proyecto es el UR3e de Universal Robots. Como se ha explicado anteriormente en los fundamentos teóricos, utilizaremos la biblioteca RTDE para de sincronizar el programa con el controlador UR sobre una conexión TCP/IP estándar, sin romper ninguna funcionalidad en tiempo real del controlador. La interfaz RTDE está disponible por defecto cuando el robot está activo, por lo que no se ha tenido que realizar ninguna configuración inicial.

Este bloque del proyecto requiere de los siguientes componentes para su correcto funcionamiento:

- Configuración de la instalación del robot, asignando una IP estática para poder conectarnos desde el ordenador estando en la misma red. A parte de la configuración e instalación de la pinza y el nuevo centro de gravedad.
- Un fichero .xml donde se pondrán los registros con los que se va a interactuar en la conexión y programa (lectura y escritura).
- Programa del robot escrito en bloques en PolyScope.

3.3.1. CONFIGURACIÓN PREVIA DEL UR3E

Se ha usado la Tablet con PolyScope proporcionada por Universal Robots, que cuenta con una interfaz gráfica de usuario y un entorno de programación, para instalar, ajustar y desarrollar el programa del robot.

Se comienza con la instalación de la herramienta, el establecimiento del punto central de la herramienta (TCP) y el cálculo del centro de masas, con la herramienta y la cámara puestas en sus respectivos sitios. En cuanto a seguridad, se ha creado un plano de protección que el robot no puede cruzar, para proteger el ordenador y el puesto de desarrollo. Para la conexión con el dispositivo externo, el ordenador, es necesario

habilitar la red y configurar una dirección IP estática, junto con la máscara de la subred (clase C) y la puerta de enlace predeterminada. No es necesario configurar el servidor DNS porque no se van a resolver nombres de dominio. Para a la conexión con el controlador UR el puerto por defecto es el 3004.

Se sigue con la configuración de entradas y salidas. Se pone un nombre a los registros para que, cuando se le asigne una acción en el programa del robot, el código sea más intuitivo. Por ejemplo, en este proyecto, se le asigna al registro de salida GP_int_out[0] un 1 cuando el programa está funcionando, y un 0 cuando esté parado. Todos los registros configurados se muestran en las siguientes tablas.

Tabla 3.1 Registros de salida utilizados

Registro	Nombre	Descripción
Tool_out[0]	GripperOFF	Desactivación del gripper
Tool_out[1]	GripperON	Activacion del gripper
GP_int_out[0]	program_running	Programa activo
GP_int_out[1]	output_i_1	Robot en movimiento

Tabla 3.2 Registros de entrada utilizados

Registro	Nombre	Descripción
GP_bool_in[126]	input_b_126	Control del gripper
GP_bool_in[127]	input_b_127	Perro guardián, programa Python
GP_float_in[0]	input_f_0	X
GP_float_in[1]	input_f_0	Y
GP_float_in[2]	input_f_0	Z
GP_float_in[3]	input_f_0	RX
GP_float_in[4]	input_f_0	RY
GP_float_in[5]	input_f_0	RZ

3.3.2. Programa Universal Robots (URP)

Utilizando PolyScope como interfaz gráfica de usuario, el URP ha seguido una programación por bloques. El programa se encarga de leer los registros de entrada para asignar valores a variables, y realizar la lógica correspondiente. También escribe en los registros de salida para mantener una comunicación bidireccional con el ordenador. De esta manera, ambos programas están comunicados y son conocedores del estado del otro.

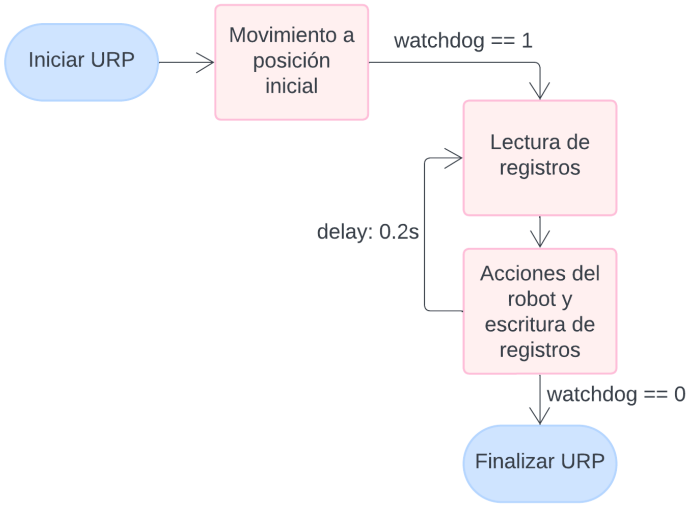

Figura 3.3 Flujograma URP

El programa UR puede dividirse en dos boques. El primer bloque se llama *AntesDeIniciar* y se ejecuta una única vez al inicio del programa. Se encarga de mover el robot a la posición inicial preestablecida y quedar en espera, hasta que el registro de estado que marca la conexión del programa Python (perro guardián) se activa (True). Esta acción da paso a la segunda parte del programa.

El segundo bloque, llamado *Programa de robot*, se encarga de leer, escribir y transformar los valores almacenados en los registros en acciones del robot. Por un lado, se utiliza la información almacenada en los registros para formar una variable combinada que define la posición a la que el robot va a moverse. Por otro lado, dependiendo del valor de otro registro, se determina el estado la pinza. También es encargado de escribir en registros de salida para indicar al sistema externo si programa está activo y si el robot está en movimiento.

Este bloque se ejecutará de forma repetida en un ciclo infinito, asegurando la realización de las acciones requeridas y actualizando su estado según sea necesario para completar eficazmente las tareas asignadas. El URP debe ponerse en ejecución antes que el programa principal para un funcionamiento correcto del sistema completo.

3.3.3. IMPLEMENTACIÓN CLASE ROBOT EN PYTHON

Se ha creado una clase para el manejo completo del robot siguiendo los principios de programación orientada a objetos, permitiendo modularidad y reutilización del código. Los atributos y métodos de esta clase están diseñados para poder cumplir con los requisitos del proyecto.

Como atributo fundamental, cuenta con una instancia de la clase RTDE que proporciona una interfaz de conexión con el robot real. Para una correcta instanciación del objeto es necesario asignar la ip, el puerto y la ubicación del archivo de configuración. Los dos primeros sirven para localizar el robot en la red y, el último, para definir, en formato .xml, el nombre, tipo y agrupación de registros que van a ser utilizados. A continuación, se explican las funciones utilizadas:

- *Connect()*: Inicia la conexión con el robot
- *Setup()*: Configura las entradas y salidas que van a ser utilizadas, apoyado en el archivo de configuración (.xml). Una vez está configurado, hace uno de la escritura de registros para indicar la pose inicial y la activación del perro guardián para indicar al programa URP que el robot está listo.

Para que el robot funcione correctamente, es necesario llamar a estos dos métodos en orden estricto. Entones, ya se pueden usar los siguientes métodos, encargados de mover el robot y controlar la pinza.

- *Move(new_pose)*: recibe como único argumento un vector de 6 elementos que refleja la pose. La función entabla conexión con el robot y lo mueve hasta alcanzar la nueva pose. Una vez alcanzada la nueva pose la ejecución de la función termina. El programa sabe que ha acabado el movimiento gracias al cambio de valor de los registros correspondientes.
- *Gripper_control(on)*: recibe como argumento una variable booleana que indica si se debe activar el gripper (True) o desactivar (False). Tiene un tiempo de espera de 1 segundo al final de la función para que no continue el programa antes de que la pinza haya cambiado de estado.

Por último, el destructor de la clase es necesario porque cambia el registro de perro guardián a cero para indicar al URP la finalizado el programa en Python o se ha destruido la instancia.

A parte de la clase principal *Robot*, se ha creado otra clase llamada *RobotException*, derivada de *Exception*, para el control de excepciones del robot.

La definición de esta clase permite gestionar el robot de manera simple y robusta desde cualquier parte del programa lo que facilita su manejo y aplicación.

3.4. IMPLEMENTACIÓN DE LA CÁMARA

La cámara utilizada ha sido la versión con enfoque automático de la OAK-D-Lite, fabricada por Luxonis. Se ha hecho uso de la biblioteca Depthai como interfaz de la conexión con la cámara desde el programa. Para que el ordenador detecte la cámara cuando se conecta es necesario instalar las dependencias. Como se dice en los fundamentos teóricos, la conexión y alimentación de la cámara van por un cable USB-C. Para el correcto desarrollo del proyecto es necesario construir un soporte para la cámara, que será comentado en el siguiente punto.

3.4.1. DISEÑO DEL SOPORTE DE LA CÁMARA

Como requisito indispensable para la realización del proyecto, era necesario un posicionamiento correcto de la cámara. Inicialmente se barajaron dos opciones. La primera era el posicionamiento en un lugar fijo externo al robot y, la segunda y definitiva, era la sujeción en el brazo robótico. Seleccionada por aportar mayor flexibilidad al poder dirigir su enfoque.

Una vez elegida la opción de sujeción en el brazo robótico, se procedió al diseño 3d del soporte, utilizando Autodesk fusión 360, que es una plataforma de software de diseño asistido por ordenador basada en la nube. El modelo creado se encuentra disponible en el repositorio de GitHub [B.1]. El soporte se fija en el último eslabón del robot, cuya forma es cilíndrica, por lo que el método de agarre es mediante una abrazadera. La abrazadera se cierra con dos tornillos, uno a cada lado. La estructura tiene forma de Y con dos pilares a los laterales que aportan estabilidad. En las esquinas de la Y hay 2 agujeros para atornillar la cámara. Esta forma minimiza el contacto con el soporte favoreciendo la refrigeración. La pieza fue impresa en plástico PLA (ácido poliláctico).

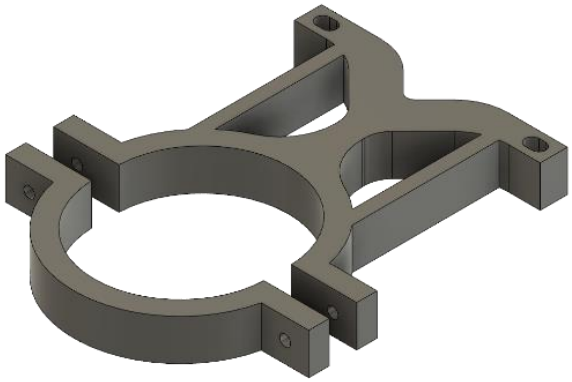

Figura 3.4 Diseño 3D del soporte

3.4.2. Implementación clase Camera en Python

Para la gestión de la cámara en el proyecto se ha creado tres clases diferentes. La primera, *CameraConfig* es la clase base y contiene los atributos de configuración: resolución, distancia focal y centro de la imagen (en pixels). La siguiente clase, denominada *CameraException*, hereda de *Exception* y se encarga de gestionar las excepciones relacionadas con la cámara.

Por último, se encuentra la clase Camera, que hereda de *CameraConfig*. Contiene los atributos y métodos necesarios para el control de la cámara en el proyecto. Como atributo principal se encuentra *pipeline*, instancia de la clase *Pipeline* de la biblioteca *Depthai*. Se encarga de la comunicación con la cámara, así como de su configuración para definir los nodos de datos que se quiere recibir. En el proyecto se han definido dos métodos diferentes para la inicialización del atributo 'pipeline' dependiendo de los datos que se quieren obtener:

- *init_rgb()*: inicialización con el nodo de la cámara central (RGB) para obtener los frames
- *init_rgb_and_pointcloud()*: inicialización con los módulos de la cámara central, las dos cámaras laterales, profundidad y nube de puntos. Más la posteriores configuraciones y enlaces para un correcto funcionamiento.

Una vez que la cámara ha sido inicializada, se han definido varios métodos adicionales para ponerla en funcionamiento. Cada uno de estos métodos tiene condiciones y opciones diferentes, que se detallan a continuación. Es importante destacar que los métodos son específicos según la inicialización que se haya realizado, ya que uno de ellos recibe menos argumentos que el otro, al necesitar menos datos.

Los métodos cuyo nombre acaba en *options* (ej. *run_with_options*), se pueden realizar acciones como recortar la imagen capturada y descargar instantáneamente al pulsar una tecla, etc. Estos métodos han sido particularmente útiles para tomar fotografías de las piezas que se utilizan en el entrenamiento de la red neuronal. El recorte de la imagen es necesario porque la red neuronal está optimizada para un tamaño de imagen específico, por lo que es conveniente obtener imágenes ya ajustadas.

Por otra parte, los métodos acabados en *condition* permiten introducir una función como argumento que será clave a la hora de decidir cuándo salir del bucle de funcionamiento del método.

Con todas las características descritas se observa la sencillez y flexibilidad de la clase para integrarse en cualquier programa. Después se explicará su aporte en el proyecto general.

3.5. SISTEMAS DE DETECCIÓN DE OBJETOS

Uno de los objetivos principales del proyecto es de un sistema de detección que permita clasificar y posicionar los objetos. El sistema se puede dividir en dos partes: identificación del objeto y adquisición de puntos clave, que se utilizara posteriormente para la adquisición de la pose del punto respecto del sistema de referencia de la cámara.

Para solucionar cada una de las partes, se han estudiado y aplicado diferentes técnicas, cada una con características y funcionamientos distintos, con el fin de descubrir que método es el más eficiente y mejor funciona en el caso concreto del trabajo. Estas técnicas se pueden dividir en tres grandes grupos: redes neuronales convolucionales, apriltags, visión artificial tradicional.

A continuación, se explican detalladamente los métodos de detección utilizados, pero antes es necesario explicar la estructura del programa, por eso mismo los dos siguientes apartados hablan sobre las clases de piezas y de detecciones.

3.5.1. CLASES DE PIEZAS

Sabiendo que las detecciones devuelven una gran cantidad de parámetros, se ha decidido crear una clase para gestionar las piezas detectadas, para facilitar la incorporación de los datos procesados en el programa. Como hay distintas detecciones, y de cada una se obtienen diferentes parámetros, se ha creado una clase base, *PieceBase*, para la agrupación de los atributos y métodos comunes, y luego una clase específica por cada tipo de detección.

Aparte, La clase *Piece* es la que agrupa todos los atributos y métodos de las anteriores, siendo la más completa. Tiene más métodos, destinados al cálculo del centro y esquinas por métodos tradicionales de visión artificial, y el cálculo de la pose. La siguiente imagen muestra el diagrama UML de las Piezas.

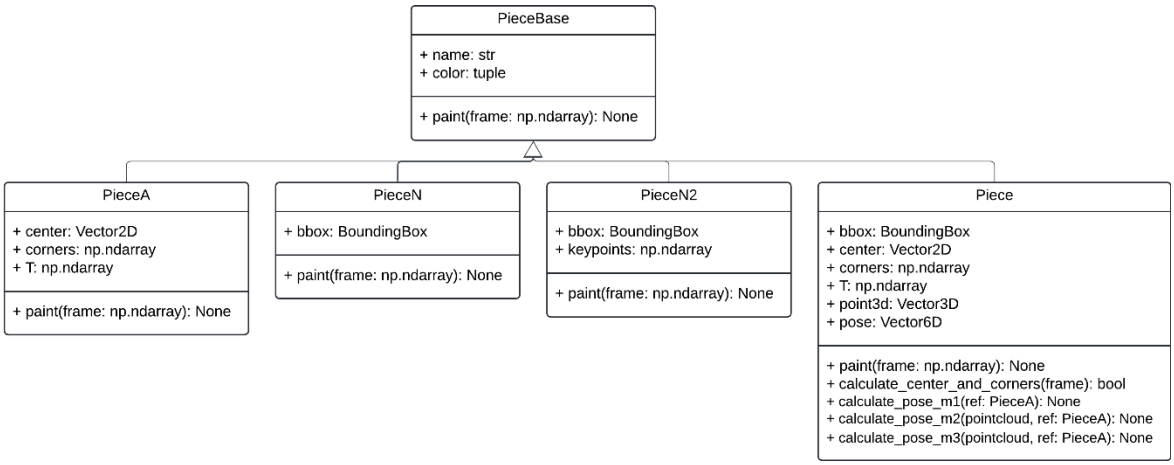

Figura 3.5 Diagrama UML de las piezas

3.5.2. CLASES DE DETECCIONES

Con el fin de que todas las detecciones compartan una estructura común, se implementa una clase interfaz denominada *DetectorInterface*.

La clase base *DetectorInterface* sirve como una interfaz genérica que define los métodos y funcionalidades esenciales que deben estar presentes en todos los detectores de objetos implementados en el trabajo.

A continuación, se muestra la representación UML de toda la estructura de detecciones que va a ser explicada posteriormente.

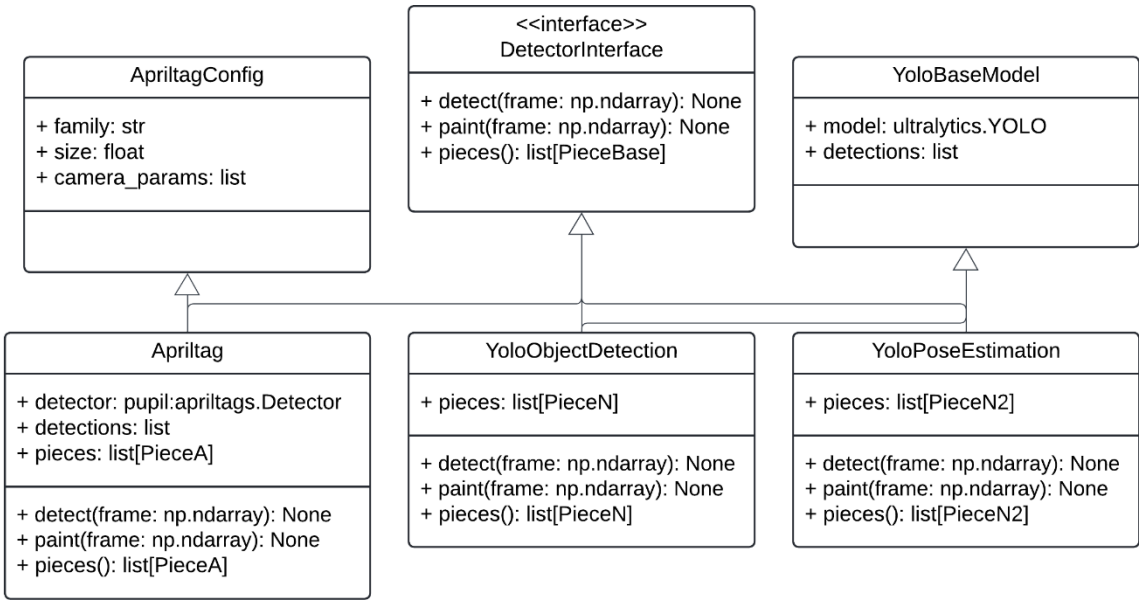

Figura 3.6 Diagrama UML de los detectores

El método *detect* sirve para identificar los objetos en la imagen dada, el segundo método se encarga de marcar en la imagen los objetos detectados, y el método *pieces* tiene como labor devolver el atributo *pieces* de las clases derivadas. Esta estructura unificada permite una comparación coherente y sistemática entre los diferentes enfoques de detección que se van a ser vistos.

3.5.3. RED NEURONAL YOLOV8

Para la detección y clasificación de objetos se ha hecho uso de la red neuronal convolucional YOLOv8. Se ha etiquetado una muestra de 300 fotos de las piezas para entrenar la red y así personificarla para detectar las piezas del trabajo. Para etiquetar las imágenes se ha utilizado la una plataforma basada en la nube, llamada *Roboflow* [10]. Esta plataforma ofrece herramientas para la anotación de datos, el preprocesamiento y la gestión de conjuntos de datos, así como la capacidad de entrenar modelos de aprendizaje profundo en tareas como detección de objetos, clasificación y segmentación de imágenes, aunque estas últimas funcionalidades no han sido utilizadas.

Gracias a la aplicación del aprendizaje por transferencia, ha sido posible afinar el modelo con muchas menos imágenes y épocas que si se hubiese entrenado de cero, logrando unos resultados aceptables en poco tiempo.

Ultralytics, empresa desarrolladora de la red, proporciona una biblioteca el Python para la gestión completa de la CNN, incluyendo entrenamiento, validación, exportación y detección.

Entrenamiento

Para el entrenamiento de la red es importante tener en cuenta que el resultado depende de los parámetros de configuración y el volumen de imágenes utilizadas. Por lo que es importante un ajuste correcto en la distribución de los datos para obtener un resultado óptimo y prometedor. Se sigue la estrategia del modelo de aprendizaje 70-20-10, que quiere decir que el 70% de los conocimientos de adquieren con la práctica del trabajo, experiencia, el 20% a partir de los resultados, otros medios, y el 10% restante mediante la evaluación independiente, sin ayuda. Esta división proporciona un equilibrio entre la cantidad de imágenes utilizadas para el entrenamiento (70% *train*), las utilizadas para la evaluación y optimización (20% *valid*) y las utilizadas para las pruebas (10% *test*). Las instrucciones de etiquetado de imágenes y la ubicación se encuentra en el archivo data.yalm.

Google Colab es una plataforma basada en la nube que permite escribir y ejecutar código en Python dentro de un entorno *Jupyter Notebook* [11]. Ofrece acceso a recursos computacionales como GPUs y TPUs, facilitando el desarrollo y entrenamiento de modelos de aprendizaje automático y análisis de datos. Como para entrenar el modelo se necesitan grandes recursos computacionales se ha decidido utilizar esta plataforma. Al utilizar la versión gratis, los recursos ofrecidos en la nube solo tienen la capacidad de entrenar la versión más pequeña de la red (yolov8n.pt), que logra no superar los requisitos mínimos de RAM para su entrenamiento. Al ser la más liviana, el entrenamiento y el análisis de imágenes es más rápido, aunque no se cuenta con la precisión de otras versiones más grandes. A continuación, se detallan los parámetros más importantes para el entrenamiento de la red.

El primer argumento de configuración del entrenamiento es la ubicación del archivo que actúa como mapa de entrenamiento, *data.yalm*. Este archivo contiene la información necesaria sobre la estructura de los datos y las rutas a los conjuntos de entrenamiento.

Las épocas de entrenamiento, *epochs*, son pasadas completas por todo el conjunto de datos de entrenamiento, durante las cuales el modelo ajusta sus pesos para mejorar. Se utilizaron 250 épocas, aunque entre 300 y 500 también es adecuado. La cantidad de épocas necesarias depende de la complejidad del problema, el tamaño y calidad del conjunto de datos, la arquitectura de la red neuronal y los hiperparámetros. Es común usar *early stopping* para detener el entrenamiento cuando el rendimiento en el conjunto de validación deja de mejorar, configurado en el argumento *patience*, que por defecto son 100 épocas.

También es posible configurar el tamaño de la imagen, i*mgsz*. 640x640 es el tamaño óptimo porque que la red y el algoritmo de entrenamiento esta optimizados para procesar este tamaño.

Por último, la ubicación del directorio donde se guardan los datos obtenidos, *Project*. Estos datos generados incluyen el mejor modelo entrenado, el último modelo, gráficas importantes como la matriz de confusión o los resultados de precisión, perdida, etc.

Clase base de detecciones YOLO

Teniendo en cuenta que hay dos detecciones con la red neuronal YOLO, se ha creado una clase base encargada de agrupar los atributos comunes de ambas detecciones. El diagrama UML de esta clase, llamada *YoloBaseModel* es el siguiente.

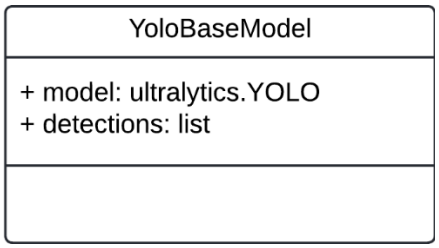

Figura 3.7 Clase base de detecciones YOLO

El primer atributo, *model*, es una instancia de la clase YOLO de Ultralytics. Almacena y gestiona todo lo referido a la red neuronal, independientemente del modelo utilizado. El segundo argumento contiene la información de los resultados de las detecciones. En los dos siguientes puntos se explican las derivadas de esta clase.

YOLOv8. Detección de objetos (*object detection*)

La clase *YoloObjectDetection* hereda de las dos siguientes clases: *DetectorInterface* y *YoloBaseModel*. Es responsable de proporcionar implementaciones concretas para los métodos abstractos de la clase interfaz. Aparte, cuenta con el atributo *pieces*, que es una lista formada por elementos de tipo *PiezaN*.

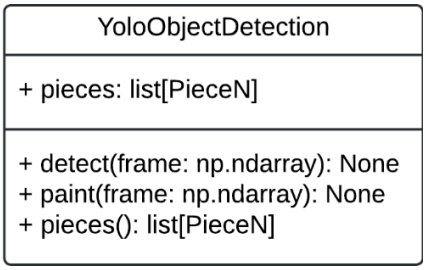

Figura 3.8 Clase YOLO detección de objetos

La clase *PiezaN* contiene la información que se adquiere en la llamada al método *detect*. También es capaz de 'pintarse' en la imagen. El método *paint* de *YoloObjectDetection* llama al método paint de cada pieza. Los atributos más importantes de la clase PieceN son la caja delimitadora (*bounding box*) y la clasificación (nombre de la pieza).

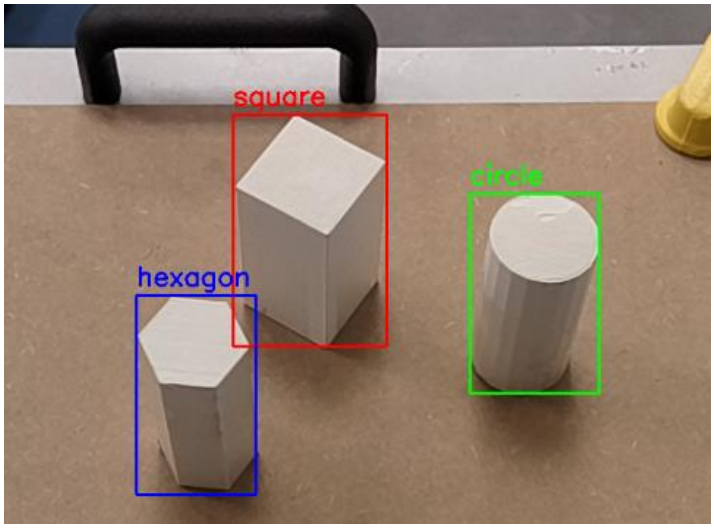

Figura 3.9 Detección de piezas con la red YOLO *object-detection*

YOLOv8. Estimación de posición (*pose estimation*)

La detección con la CNN de YOLO, con el modelo estimación de posición, es capaz de adquirir todos los argumentos de la red YOLO *object-detection*, más una serie de puntos característicos con los que previamente se ha entrenado la red. Al adquirir más información, el tipo de piezas de esta clase es *PieceN2*. Este tipo de pieza cuenta con un atributo más, llamado *keypoints*, que contiene una lista de vectores de 2 dimensiones para almacenar los puntos clave del objeto en la imagen.

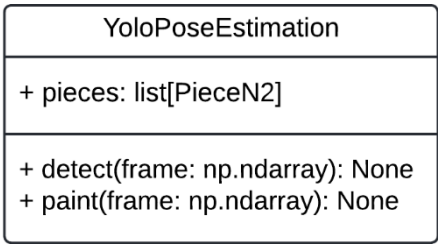

Figura 3.10 Clase YOLO estimación de pose

Figura 3.11 Detección de una pieza con la red YOLO *pose-estimation* con los puntos clave en azul

3.5.4. APRILTAGS

Los apriltags proporcionan, de una manera muy eficiente y precisa, la posición y orientación del apriltag respecto de la cámara. Como se puede deducir, es otro tipo de detección, por lo que la clase *Apriltag* hereda los métodos de la interfaz. El detector utilizado se encuentra en la librería pupils_apriltag. Hay que tener en cuenta que, para la utilización de este método, es necesario que la cámara esté calibrada y conocer los parámetros de distancia focal (fx,fy) y centro de la imagen (cx,cy), ambos dependientes de la resolución. También, es necesario conocer la medida del lado del cuadrado, 15mm, y la familia, Tag36h11. Hay una clase específica, *ApriltatgConfig*, creada para el almacenamiento de todos estos parámetros.

Para asegurar una detección precisa, los apriltags deben estar adheridos a una superficie plana, en el proyecto se encuentran en la cara de las piezas. Sin embargo, esta característica también puede ser considerada como una desventaja en ciertas situaciones. Por ejemplo, los apriltags deben ser visibles para que puedan ser detectados correctamente por la cámara. Además, si estás analizando un objeto pequeño o irregular, o si no puedes alterar su estética, entonces no se puede hacer uso de esta tecnología. Esto limita la aplicabilidad de los apriltags en ciertos contextos donde la colocación de etiquetas no es posible o práctica.

El tipo de piezas detectadas es *PieceA*. La clase *PieceA*, hereda de *PieceBase*, y añade los siguientes atributos: el centro del apriltag, las esquinas del apriltag y, la mayor ventaja de este detector, la matriz de transformación al sistema de referencia de la cámara. A pesar de que los apriltags tienen identificador, no se ha hecho uso para la clasificación de la pieza, únicamente para saber el apriltag de referencia, explicado posteriormente.

La clase *Apriltag*, aparte de heredar los métodos de la interfaz, hereda los atributos de la clase *ApriltagConfig*. El diagrama UML es el siguiente.

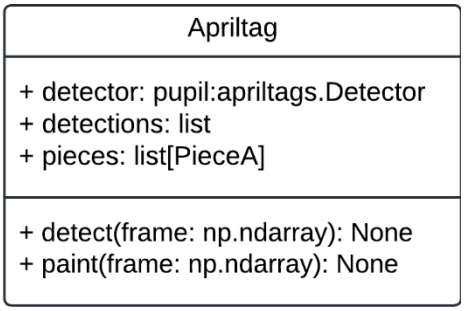

Figura 3.12 Clase Apriltag

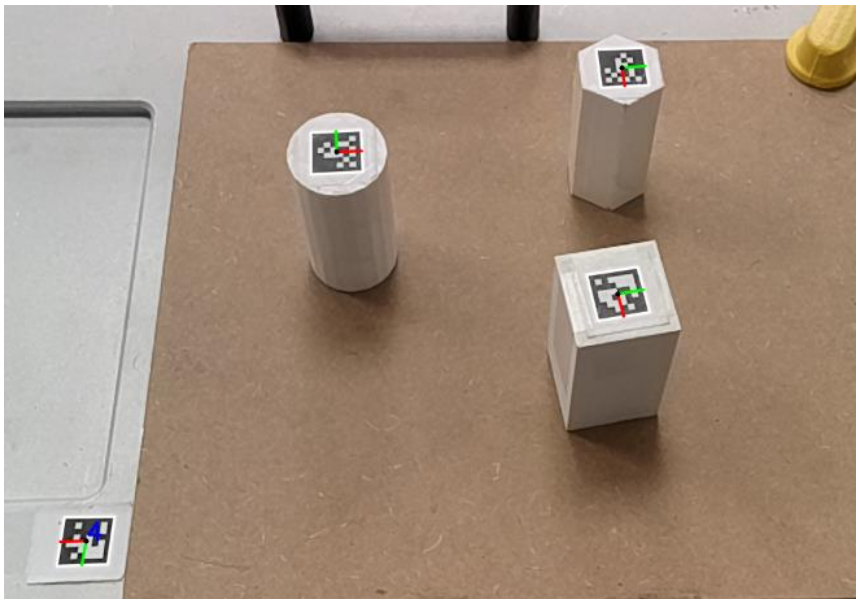

Figura 3.13 Detección de apriltags

3.5.5. VISIÓN ARTIFICIAL CLÁSICA

Al no adquirir buenos resultados con la detección mediante YOLO *pose-estimation,* debido a la baja precisión en la detección de puntos característicos, se optó por un análisis clásico mediante la utilización de filtros y algoritmos. La falta de precisión en la detección se debió a varios factores: la escasez de imágenes etiquetadas disponibles para entrenar el modelo, el poco tiempo dedicado al entrenamiento y la naturaleza simétrica de las piezas, que dificultaba aún más el proceso de entrenamiento. Estos desafíos llevaron a la implementación de métodos más tradicionales y robustos para el procesamiento de imágenes, como el uso de filtros específicos y algoritmos establecidos, que no dependen de grandes volúmenes de datos de entrenamiento ni son sensibles a las simetrías presentes en las piezas. Con este análisis se obtiene el centro

y lado de las piezas, suficiente para, con la ayuda de la nube de puntos, obtener la posición y rotación de la pieza respecto de la cámara, calculando así la matriz de transformación.

Este detector no está definido como una clase, ya que es específico del tipo de pieza. Sus métodos se encuentran definidos en la clase pieza general, *Piece*, y las funciones generalizadas se encuentran en el archivo *helper_functions.py*. La biblioteca OpenCV ha sido fundamental para el manipulación, análisis y procesamiento de imágenes y videos en tiempo real [18].

Antes de aplicar estos métodos, se detecta la pieza, y se recorta la imagen adquiriendo únicamente la sección de la imagen marcada por la caja delimitadora de la pieza. Posteriormente se convierte a escala de grises, para simplificar el procesamiento y mejorar la precisión de los algoritmos.

Para las piezas con lados, se aplica un filtro bilateral para reducir el ruido de la imagen sin difuminar los bordes.

Después, para la detección de esquinas, se utiliza el algoritmo de Harris [12], que obtiene un valor por cada píxel de la imagen, indicando la probabilidad de ser una esquina. El algoritmo se ha configurado con los siguientes valores:

- *block_size* (8): Tamaño del vecindario considerado para la detección de esquinas.
- *ksize* (3): Tamaño del kernel[1] utilizado para el cálculo de la derivada de Sobel.
- *k* (0.05): Parámetro libre de Harris que determina la sensibilidad del detector.

Posteriormente, se aplica un umbral de 0.01 para seleccionar solo los píxeles con valores altos, identificando así las esquinas significativas. A continuación, los píxeles detectados como esquinas se agrupan si están cerca y se calcula el centroide de cada grupo, para obtener un único punto representativo de cada esquina. Este proceso permite identificar esquinas de manera precisa y eficiente en las imágenes de las piezas analizadas.

[1] Kernel: matriz que es utiliza para realizar operaciones sobre una imagen mediante convolución.

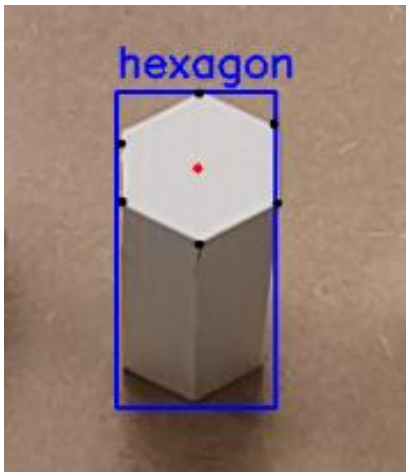

Figura 3.14 Pieza con esquinas detectadas

Para detectar las elipses en la imagen, se ha desarrollado un método que sigue varios pasos específicos. Primero, se aplica un filtro de mediana para reducir el ruido en la imagen original que, a diferencia del filtro bilateral, potencia los contornos circulares. Para la aplicación del filtro se ha utilizado un kernel de 11 píxeles.

Luego, se utiliza el algoritmo de Canny [21], con un umbral inferior de 80 y un umbral superior de 200, que detecta bordes encontrando áreas de cambios bruscos en la intensidad de la imagen, después de aplicar un suavizado inicial. Utiliza umbrales para identificar y conectar estos bordes en la imagen definitiva.

Posteriormente, se encuentran los contornos en la imagen de bordes y se itera a través de ellos para identificar el contorno que se encuentra más arriba en la imagen. Este contorno corresponde, potencialmente, a la cara superior del cilindro. Se verifica que el contorno seleccionado tenga al menos cinco puntos, requisito necesario para ajustar una elipse. Finalmente, se ajusta una elipse a este contorno utilizando la función *cv2.fitEllipse*, proporcionada por OpenCV, de la cual se obtienen todos los datos necesarios para poder representarla (centro, lados y ángulo).

Figura 3.15 Pieza con elipse detectada

Es fundamental considerar que la iluminación de las piezas, así como la diferencia de color entre la pieza y el fondo, son factores cruciales para el éxito de estos métodos de detección. Todo influye en el resultado final, por lo que se han establecido unos valores de ajuste específicos que corresponden con las condiciones de luminosidad del laboratorio, el color de la pieza y el color del fondo. Fuera de este entorno controlado, sería necesario ajustar todos los valores de los filtros y algoritmos definidos anteriormente para mantener la precisión y eficiencia de la detección.

3.6. INTEGRACIÓN DEL SISTEMA COMPLETO

Para resolver el problema de detección y posicionamiento de piezas, se han desarrollado varios métodos, cada uno con su propio enfoque. A pesar de ello, todos los sistemas comparten una estructura común y una interacción similar entre clases, lo que permite utilizar un flujograma general para todos los métodos. Esta sección describe cómo se integran y funcionan los diferentes componentes del sistema.

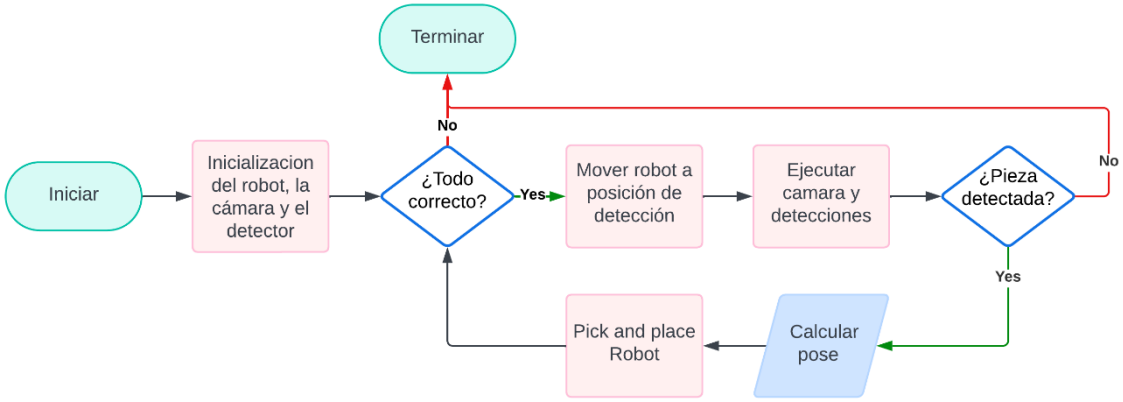

Figura 3.16 Flujograma del sistema completo

El programa principal coordina la interacción entre el robot, la cámara, la detección de objetos y otras funcionalidades necesarias. El proceso se inicia con la llamada a la función *main*, que instancia los objetos necesarios para el sistema. Estos incluyen el robot, la cámara, el apriltag y la red neuronal (ya sea *nn_od_model*[2] o *nn_pose_model*[3], dependiendo del sistema implementado).

La inicialización de los componentes es crucial. La cámara tiene dos modos de inicialización: *RGB* y *RGB_pointcloud*. El robot también se inicia y conecta. Si ocurre algún

[2] nn_od_model: red neuronal convolucional de tipo detección de objetos
[3] nn_pose_model: red neuronal convolucional de tipo estimación de pose

problema durante la inicialización, como la desconexión de un componente, se captura la excepción y se establece un variable *flag* con valor booleano *False*, que provoca la finalización del programa. El manejo de excepciones es vital, ya que cualquier desconexión o mal funcionamiento puede causar fallos significativos en el sistema.

Una vez que la inicialización es exitosa, se ejecuta el bloque de detección y manipulación de piezas. Este proceso se encuentra dentro de un bucle controlado por el *flag*. Dentro del bucle, se llama a una función estática de la clase *Coordinador*, que recibe las instancias y maneja todo el proceso de movimiento del robot, detección de la pieza, cálculo de pose y el *pick and place*. Dependiendo del valor booleano que devuelva la función, asignado a *flag*, el bucle se repite o el programa finaliza.

Además, se realiza la transformación de las coordenadas de detección a un sistema de referencia 3D, estimando la pose de la pieza con la ayuda de la cámara y transformando estas coordenadas al sistema de referencia de la base del robot. Esto se logra utilizando una referencia conocida, un apriltag, que está pegado en una zona específica del soporte del robot. La ubicación del apriltag es conocida en relación con la base del robot, lo que permite realizar la transformación de coordenadas desde el sistema de la cámara al sistema del robot de manera precisa.

En resumen, el sistema se integra de tal manera que, tras una exitosa inicialización, el programa principal coordina todos los componentes necesarios para detectar, posicionar y manipular las piezas de forma eficiente, utilizando referencias conocidas para asegurar la precisión en el cambio de sistemas de referencia.

3.6.1. PICK AND PLACE

La clase coordinador tiene definido un método estático, llamado *combined_movement*, encargado de realizar el *pick and place* del sistema. Recibe como argumentos de entrada la instancia del robot, la pieza y la tolerancia del sistema. Hay tres tipos distintos de tolerancia, dependiendo del agujero donde se quiere dejar la pieza.

Como se puede suponer, este método adquiere la pose de la pieza y la pose del agujero donde va colocada. Una vez se tiene las dos poses, se ejecuta una secuencia de movimientos del robot y de la pinza para realizar la satisfactoriamente. El proceso termina volviendo a la posición inicial. Si hay algún problema en la ejecución de los movimientos, el robot puede detenerse desde el URP o desde el programa principal.

Cuando finaliza el método termina la función principal del coordinador y se vuelve al main, que este, a su vez, si el *flag* es verdadero vuelve a llamar a la función principal de nuevo, y si es falso, finaliza el programa.

3.6.2. CÁLCULO DE LA POSE DE LA PIEZA

Al haber desarrollado distintos sistemas de detección, es necesario complementarlo con un sistema que calcule la pose del objeto respecto de la cámara y, después, respecto de la base del robot. Cuando se habla de la pose del objeto significa la posición y orientación del objeto respecto de un sistema de referencia.

Con el sistema de detección mediante apriltags, no es necesario calcular la pose del objeto, ya que el propio sistema entrega la matriz de transformación del objeto respecto de la cámara. Con esta matriz se conoce la pose de la pieza.

Con los demás sistemas es necesario calcular la pose. Se parte del conocimiento de centro y lados en la imagen en 2d, adquiridos con la CNN *pose-estimation* o con métodos de visión artificial tradicionales. Para este cálculo de pose se utiliza la nube de puntos correspondiente a la imagen. Gracias a ella se adquiere el análogo del centro de la pieza en 3d. Se ha utilizado la librería Open3D, la cual ofrece herramientas avanzadas para la visualización, manipulación y procesamiento de nubes de puntos, mallas y geometría 3D [17].

Para calcular la rotación de la pieza respecto del apriltag, se utiliza la imagen 2d. Se forman dos vectores, el primero formado por la unión de dos vértices consecutivos de la cara de la pieza, y el segundo determinado por eje X del apriltag. Una vez se tienen los vectores se calcula el ángulo que forman entre ellos. Esto es posible porque la cámara se encuentra en una posición casi cenital, permitiendo la aproximación de que ambos vectores se encuentran en el mismo plano. La rotación se produce entorno al eje Z. Sabiendo esto se adquiere la matriz de rotación. Una vez con la rotación y posición respecto de la cámara, ya se tiene la pose y, por consiguiente, la matriz de transformación.

Una vez se tiene la pose de la pieza respecto de la cámara es necesario adquirir la pose respecto de la base del robot, ya que el robot se maneja desde ese sistema de referencia. Para lograr en cambio de sistemas se hace uso de un apriltag de referencia, que se encuentra situado en un lugar visible y conocido dentro del área de detección de piezas. Gracias a este apriltag, podemos hacer las transformaciones pertinentes, del sistema de referencia de la cámara, al sistema del apriltag, y del apriltag al de la base del robot. Estos cambios de sistema se hacen con matrices de transformación homogénea.

3.6.3. SISTEMAS DE DETECCIÓN Y POSICIONAMIENTO POSIBLES

Con el software desarrollado, es posible crear diferentes sistemas de detección y posicionamiento en función de los métodos utilizados. A continuación, se muestra una imagen con las posibles combinaciones.

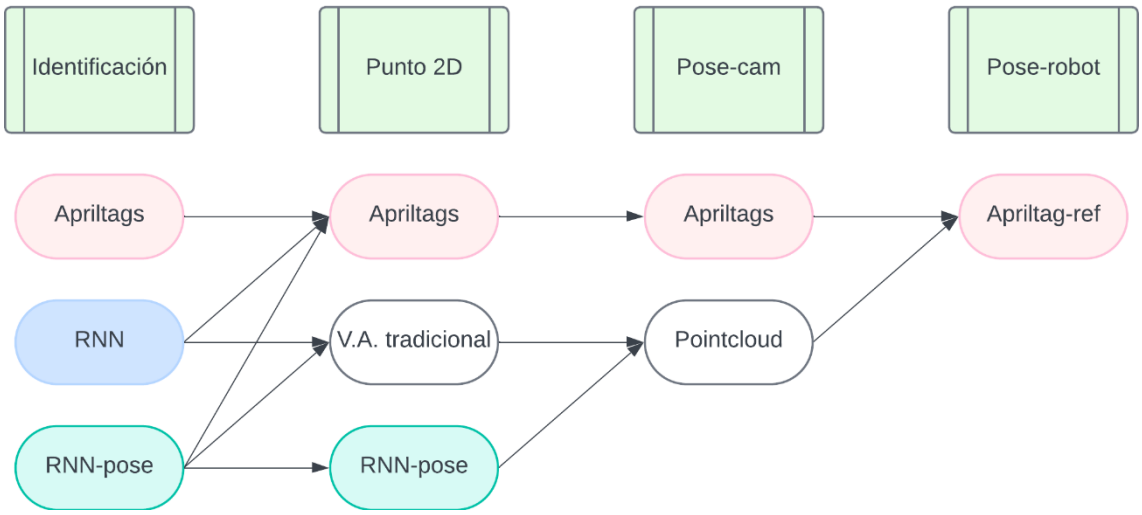

Figura 3.17 Posibles combinaciones de sistemas

Observando el diagrama anterior, hay una gran variedad de sistemas que pueden ser creados. Si se utiliza la detección con apriltags, se adquieren todos los datos necesarios para adquirir la pose de la pieza respecto de la cámara. En cambio, utilizando la RNN, es necesario la detección del apriltag para conocer la pose del objeto o, aplicar visión artificial tradicional y la nube de puntos para llegar al mismo resultado. Por último, aplicando la RNN *pose-estimation*, se pueden seguir diferentes pasos para llegar al mismo resultado. Por un lado, puede seguir los pasos de la RNN, ya que posee los mismos datos, y por otro lado, utilizando los keypoints de la red para detectar centro y esquinas, solo habría que aplicar la estimación de pose con la nube de puntos.

Una vez se tiene la pose de la pieza respecto de la cámara que, como se acaba de ver, es posible adquirirla de muchas maneras, es necesario pasar al sistema de referencia de la base del robot. Para ello es necesario pasar por el apriltag de referencia, el cual sirve de nexo entre la cámara y la base del robot.

En el trabajo, se decide realizar tres sistemas distintos, con el fin de abarcar los máximos métodos posibles.

El primer sistema identifica las piezas con la red neuronal YOLO previamente entrenada. Posteriormente, Se obtiene el punto 2d del centro de la pieza y la pose respecto de la cámara con el apriltag que lleva pegado en la cara superior y, por último, se obtiene la pose del objeto respecto de la base del robot, utilizando el apriltag de referencia.

El segundo sistema identifica y obtiene los puntos característicos de la pieza utilizando la red neuronal con estimación de posición. Después, se estima la pose de la pieza respecto de la cámara utilizando la nube de puntos generada y el apriltag de referencia. Una vez se tiene la pose respecto de la cámara, se utiliza el apriltag de referencia como en el anterior método para pasar al sistema de referencia de la base del robot.

El tercer sistema utiliza la red neuronal para detectar y clasificar el objeto. A continuación, obtiene los puntos clave aplicando métodos de visión artificial tradicional y, después, se siguen los mismos pasos que en el segundo sistema.

En los siguientes apartados se describen profundamente los sistemas.

3.6.4. SISTEMA 1: RNN | APRILTAGS | APRILTAG-REF

Este método de resolución del problema es el más sencillo. Se ha utilizado la red neuronal YOLO para la detección de objetos, y los apriltags para conocer la posición y orientación respecto de la cámara. Los objetos, al tener en la cara superior un apriltag cada uno, se puede conocer fácilmente la pose de la cara superior respecto del sistema de referencia de la cámara. Al tener dos detectores distintitos, la RNN y los apriltags, es necesario un coordinador de detecciones que combine los diferentes tipos de piezas detectadas, con el fin de adquirir la máxima información posible.

El proceso de combinar los apriltags con la caja delimitadora es por pertenencia, si el centro de un apriltag se encuentra dentro de la caja delimitadora de una pieza, entonces ese apriltag se asigna a esa pieza (pasaba a formar parte de ella). Se crea una nueva pieza, tipo *Piece*, que guarda tanto la información del apriltag como el de la *PiezaN*. Gracias a la librería apriltag, se tiene la matriz de transformación del centro del apriltag a la cámara, por lo que solo falta una manera de pasar del sistema de referencia de la cámara al sistema de referencia de la base del robot. Este problema se soluciona utilizando un apriltag de referencia que se encuentra pegado en un lugar visible y conocido en la zona de detección de objetos.

Se conoce la posición y orientación del apriltag de referencia respecto al sistema de referencia de la base del robot, por lo que se tiene matriz de transformación de un sistema al otro.

Con todo lo adquirido anteriormente, ya es posible pasar del sistema de referencia de la cámara al sistema de referencia del robot.

Una vez se tiene la pose del objeto respecto del robot, se lanza la función del coordinador que realiza el *pick and place*, donde se ejecutan una serie de instrucciones que van desde coger la pieza detectada hasta dejarla en el hoyo correspondiente.

3.6.5. SISTEMA 2: RNN-POSE | POINTCLOUD | APRILTAG-REF

A diferencia del primero, la red entrenada YOLO es de tipo estimación de pose, por lo que detecta también puntos clave. Los puntos clave entrenados son los vértices y el centro de la cara superior de la pieza.

Una vez realizada la detección, se utilizaba la nube de puntos generada por la cámara para adquirir el análogo del centro de la pieza en 3d.

Una vez se tiene el punto en 3d falta la rotación de la pieza respecto del apriltag de referencia. Se calcula cogiendo el lado creado por 2 vértices consecutivos, en el proyecto se cogen los dos puntos superiores, y el lado del eje x del apriltag. Como la cámara está en una posición casi cenital, se puede aproximar que el apriltag y la cara superior de la pieza se encuentran en el mismo plano, se calcula la diferencia de grados entre los dos vectores formados en la imagen.

Con el punto 3d y la rotación se forma la matriz de transformación de la pieza a la cámara. Todos los pasos posteriores son iguales que en el primer sistema.

3.6.6. SISTEMA 3: RNN | V.A. CLÁSICA | POINTCLOUD | APRILTAG-REF

Al no lograr una clara detección de puntos clave con la red neuronal *pose-estimation*, se procedió a la detección de vértices y contornos con métodos tradicionales de visión artificial, explicados anteriormente.

La parte inicial del programa es igual que en el sistema 1, detectar objetos con la CNN. Después de ello, se recorta la imagen por la caja delimitadora de la pieza, y se aplican los métodos tradicionales de visión artificial para obtener en centro y los lados de las piezas, en el caso del círculo el centro y las medidas de la elipse.

Se adquiere el análogo del centro de la pieza en 3d utilizando la nube de puntos y la rotación se consigue aproximando la rotación de los vectores de un lado de la pieza con el eje X del apriltag de referencia, igual que en el sistema 2. Una vez se tiene la matriz de transformación, se procede con el *pick and place*.

Capítulo 4. RESULTADOS

En este capítulo se presentan y se analizan los resultados obtenidos en las diferentes áreas del sistema, abarcando tanto la clasificación y detección de objetos como el proceso de *pick and place* del robot. El objetivo principal es evaluar la efectividad y precisión de los algoritmos desarrollados, comprobando la capacidad del sistema para realizar las tareas asignadas de manera eficiente y confiable.

Otro aspecto importante que se abordará en este capítulo es la evaluación de las dificultades y desafíos enfrentados a lo largo del proceso de desarrollo e implementación del sistema. Estas dificultades pueden incluir problemas técnicos, limitaciones de hardware y software, y desafíos relacionados con la integración y sincronización de los distintos componentes del sistema. Al identificar y analizar estas dificultades, se pretende proporcionar una visión más completa del proceso y ofrecer recomendaciones para futuras mejoras, vistas en el siguiente capítulo.

4.1. RESULTADOS DE ENTRENAMIENTO DE LA RED NEURONAL

En esta sección se presentan y analizan los resultados obtenidos durante el proceso de entrenamiento de la red neuronal YOLOv8, para tareas de detección de objetos y estimación de pose. El proceso de entrenamiento se lleva a cabo utilizando un modelo preentrenado basado en conjuntos de datos COCO, y un conjunto de imágenes de las

piezas utilizadas en el proyecto con variaciones en la iluminación y entorno. Estos datos han sido etiquetados para proporcionar información sobre las piezas. Hay dos grupos de imágenes etiquetadas, uno para cada red neuronal. La diferencia principal es que para la estimación de pose es necesario el etiquetado con puntos clave, no necesario para la detección de objetos. La evaluación del modelo se realiza utilizando un conjunto de datos de validación independientes, con los que la red no ha sido entrenada. A continuación de muestran los resultados de los diferentes entrenamientos realizados.

4.1.1. DETECCIÓN DE OBJETOS

El modelo preentrenado ha sido la versión *yolov8n.pt*. Es la versión de la red más pequeña que se ofrece. Este modelo se ha entrenado con 300 imágenes, durante 300 épocas, implicando un tiempo de procesamiento de 8 horas y media. Como resultado final se obtiene el modelo con mejores resultados que antes del entrenamiento. El optimizador de la etapa final es eliminado para reducir el tamaño del modelo.

En cuanto a la configuración del sistema, La validación se realizó en un entorno con Python 3.10.12 y PyTorch 2.3.1, usando una CPU Intel Core i5-10500 a 3,10GHz. El modelo tiene 168 capas, con un total de 3.006.233 parámetros y 8,1 GFLOPs (Operaciones de Punto Flotante por Segundo).

En la Figura 4.1 Métricas de entrenamiento YOLO *object detection*, se muestran los gráficos de evolución a lo largo de las épocas, tanto para el conjunto de entrenamiento, *train*, como para el conjunto de validación, *val*.

La red tiene un rendimiento muy alto en la detección de objetos, con métricas de precisión[4] y exhaustividad[5] (*precision and recall)* superiores al 95%. Estas gráficas tienen tendencia asintótica a 1, que es el valor perfecto. Indica que el algoritmo de optimización converge, el error disminuye a medida que las iteraciones aumentan.

Las pérdidas en localización (*box_loss*) y clasificación (*cls_loss*) se encuentran por debajo del 0,2, buenos resultados, y calidad de predicciones (distribución focal, *dfl_loss*) por debajo de un 0,8, un número aceptable. Todas las pérdidas cuentan también con una gráfica asintótica, pero con tendencia a 0, ya que indican el error.

[4] Precisión: proporción de verdaderos positivos (TP) entre el total de verdaderos positivos y falsos positivos (FP).
[5] Exhaustividad: proporción de verdaderos positivos entre el total de verdaderos positivos y falsos negativos (FN).

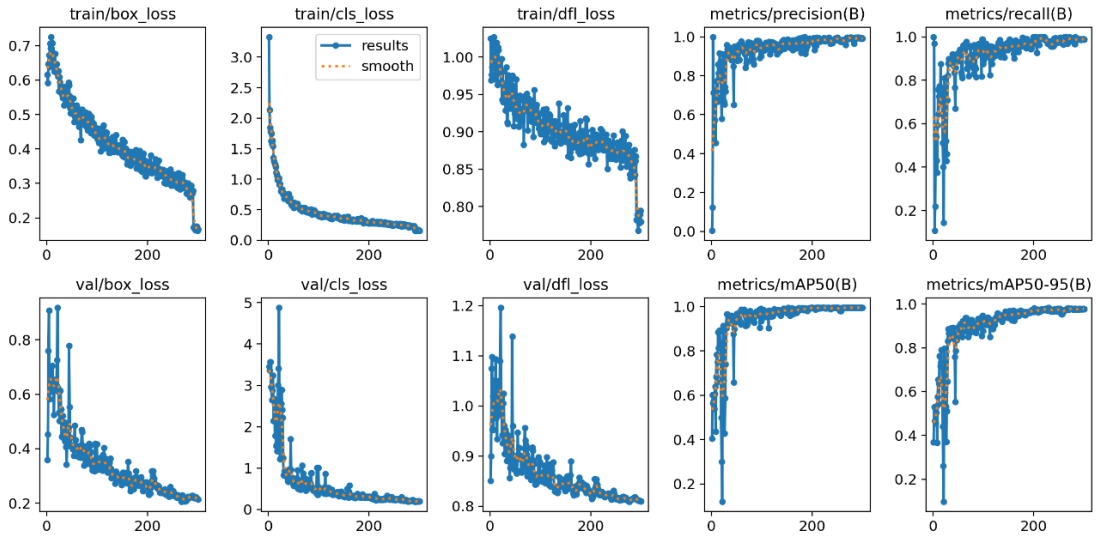

Figura 4.1 Métricas de entrenamiento YOLO *object detection*

La matriz de confusión es fundamental para evaluar rendimiento del modelo de clasificación. Se puede ver como la clasificación del cuadrado ha sido perfecta. El círculo ha sido confundido con el fondo y con el hexágono una vez. A su vez, el hexágono ha sido confundido con el fondo dos veces y con el círculo una vez. Hay que tener en cuenta que estos resultados se generan con el conjunto de validación. A continuación, se muestra la gráfica obtenida.

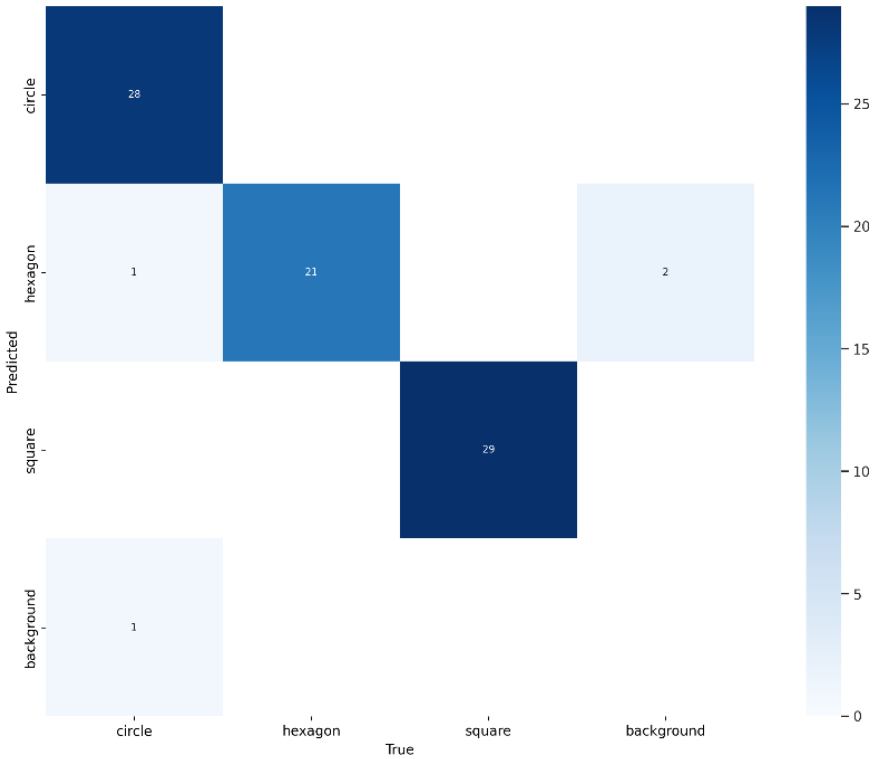

Figura 4.2 Matriz de confusión

4.1.2. ESTIMACIÓN DE POSE

El modelo YOLOv8 de estimación de pose fue entrenado utilizando un conjunto de datos específicamente etiquetado para la identificación de puntos clave en una pieza concreta, como se puede ver en la Figura 4.3.

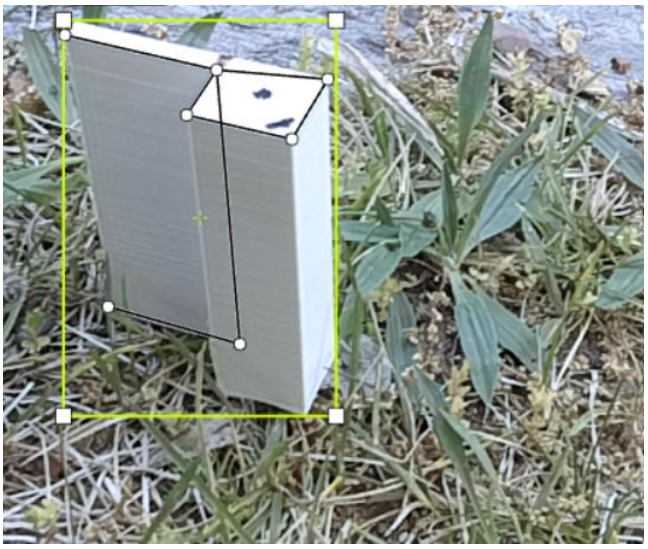

Figura 4.3 Pieza etiquetada con puntos clave

La versión concreta ha sido *yolov8s-pose.pt*. Es la segunda versión más pequeña ofrecida. Este modelo se ha entrenado con 181 imágenes, durante 450 épocas, implicando un tiempo de procesamiento de 31 horas. Como resultado final se obtiene el modelo con mejores resultados. El optimizador es eliminado para reducir el tamaño del modelo.

En cuanto a la configuración del sistema, La validación se realizó en un entorno con Python 3.11.4 y PyTorch 2.2.1, usando una CPU Intel Core i7-8550U a 1,8GHz. El entorno de entrenamiento de este modelo ha sido distinto al anterior, porque requería más recursos computacionales por su mayor complejidad. El modelo final consiste en 187 capas, con un total de 11.413.938 parámetros y 29,4 GFLOPs, lo que refleja su complejidad y capacidad de procesamiento.

Los resultados de la validación del modelo se resumen en varias métricas clave, que se muestran en gráficas y desglosan a continuación.

A parte de las pérdidas que coinciden con el otro modelo, que tienen unos valores por debajo del 0,5 en box_loss y cls_loss, y un 0,9 en dfl_loss, se tienen dos perdidas más, la de posición (*pose_loss*) y la de presencia de objetos en zonas de interés (kobj_loss). La primera es buena, alrededor del 0,5, y la segunda también, estando por debajo del 0,2. Hay una diferencia clara entre las gráficas de *train* y las de *val*. La tendencia más lineal del conjunto *train* indica que el modelo puede estar sobreajustando

los datos de entrenamiento. Esto quiere decir que se está adaptando demasiado a los datos específicos y es probable que no generalice bien los nuevos datos.

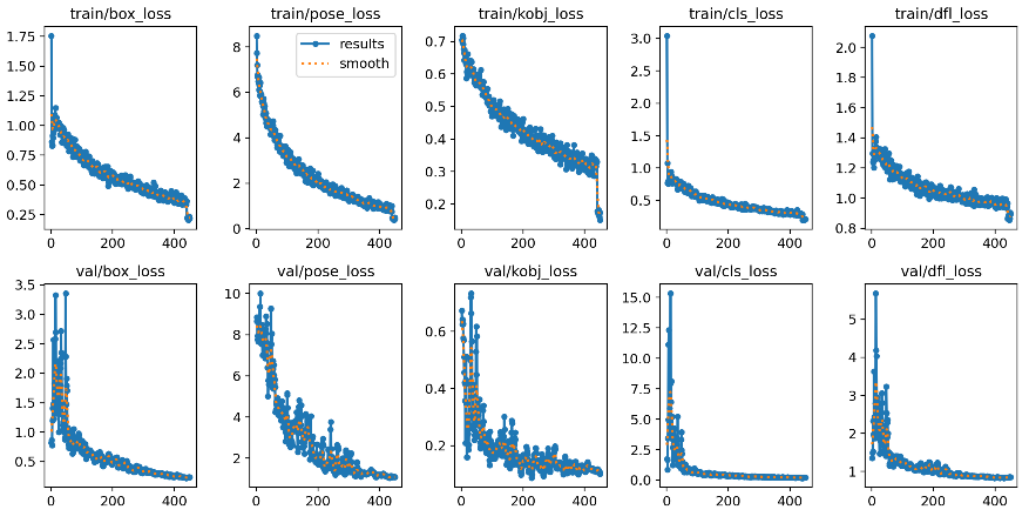

Figura 4.4 Gráficas de perdidas YOLO *pose-estimation*

Una vez vistas las gráficas de pérdidas pasamos a ver las métricas de precisión y exhaustividad, tanto para la detección de cajas (B), como para la estimación de pose (P). Se observa que cumplen con más de un 90% en ambas, lo que significa que el modelo funciona bien en estos términos. Estas gráficas tienen formas asintóticas tendiendo a 1, pero tardan más épocas en alcanzar el límite que en la otra red neuronal. En especial la métrica mpAP50-95, que tiene un comportamiento más lineal. Al terminar el entrenamiento, no alcanza el 90%. Esto significa que la métrica no ha alcanzado un valor estable o máximo y continúa cambiando con el número de iteraciones.

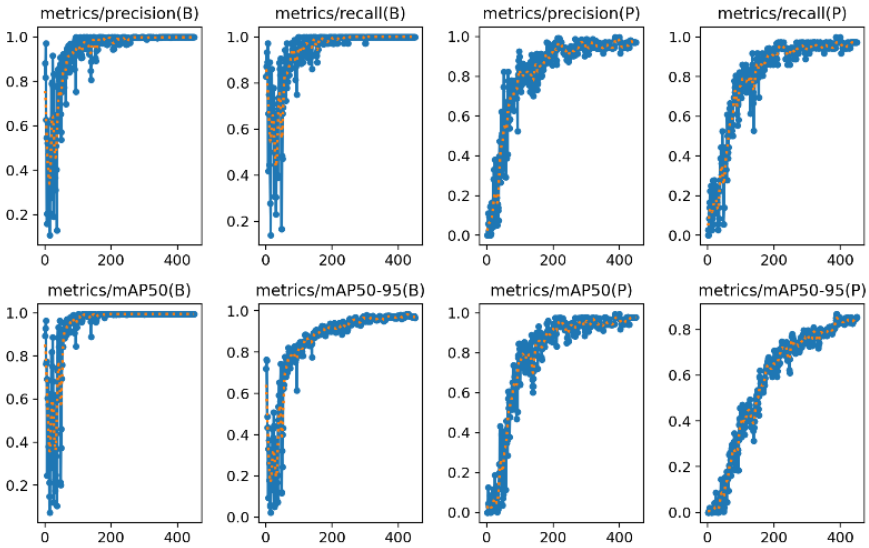

Figura 4.5 Métricas de entrenamiento YOLO *pose-estimation*

Además de las métricas anteriores, se dispone de la matriz de confusión, cuya dimensión es menor en este modelo debido a que ha sido entrenado únicamente con una clase de pieza. Al no haber ningún parámetro fuera de la diagonal principal, significa que se han clasificado correctamente todas las instancias de esta categoría, mostrando una precisión perfecta. Es importante destacar que se han utilizado exclusivamente los datos de validación, equivalentes al 20% del conjunto total de datos, que en este caso específico consta de 36 imágenes.

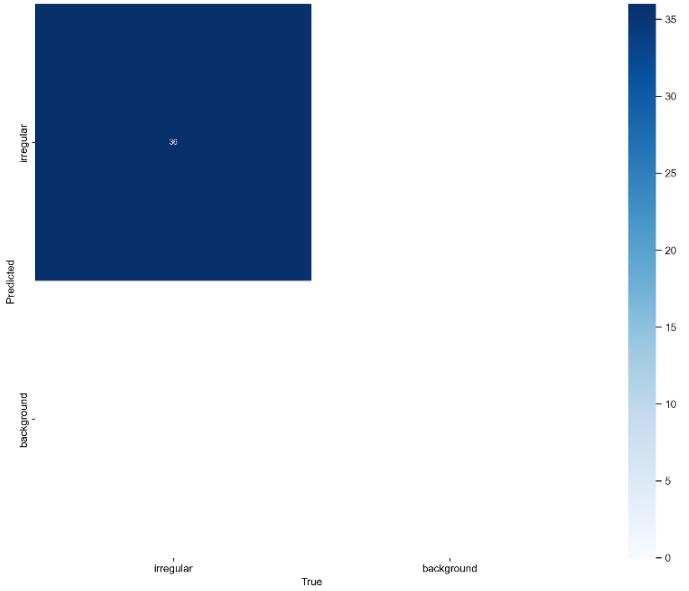

Figura 4.6 Matriz de confusión

Sin embargo, a pesar de los resultados aceptables obtenidos en la evaluación inicial del modelo, su incapacidad para adaptarse y generalizar adecuadamente a nuevas situaciones y entornos, debido a las limitaciones en el conjunto de datos de entrenamiento, ha llevado a su descarte en la creación del sistema completo. Como se puede ver en la siguiente imagen, la red no cumple con la precisión necesaria en el marcado de puntos clave.

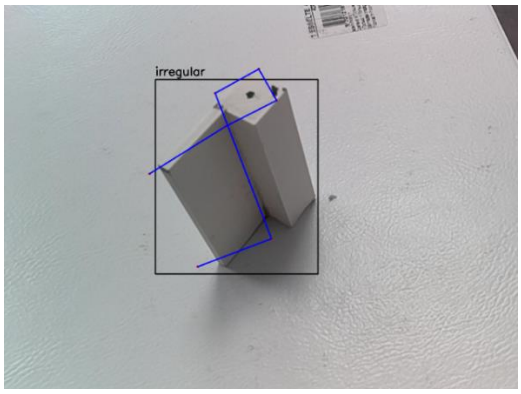

Figura 4.7 Detección de la pieza con el modelo entrenado con los puntos clave calculados por la red neuronal pintados sobre la pieza en azul

Esto destaca la importancia de contar con conjuntos de datos amplios y representativos para el entrenamiento de modelos de aprendizaje automático, especialmente en aplicaciones críticas donde se requiere un alto nivel de precisión y robustez.

4.2. RESULTADOS DEL SISTEMA

En este apartado se van a analizar los resultados obtenidos de los sistemas al completo, dependientes de detección, la estimación de la pose y el *pick and place* del robot. Para medir el error cometido en el sistema completo, El tablero de los hoyos donde hay que dejar las piezas tiene diferentes precisiones, permitiendo más o menos tolerancia. Cada pieza tiene tres agujeros, con un tamaño de un 5%, un 10%, y un 15%. Se han llevado a cabo 10 iteraciones por cada pieza por cada tolerancia, por lo que al final hay un total de 90 iteraciones por cada sistema. En cada etapa, se puntúa con un 1 si hay acierto y con un 0 si hay fallo. El fallo en una etapa conlleva, automáticamente, al fallo en las sucesivas.

Se han realizado dos tablas de resultados finales diferentes por cada sistema. La primera tabla muestra los éxitos sobre las iteraciones completas, es decir, el error de una etapa repercute en todas las demás. La otra tabla muestra el éxito de cada etapa cuando las anteriores han funcionado correctamente, por lo que no tiene en cuenta el error anterior, sino únicamente el que se comete en la etapa. Las tablas de resultados completas se encuentran en Anexo C.

4.2.1. SISTEMA 1: RNN | APRILTAGS | APRILTAG-REF

En el sistema 1 se ha utilizado la CNN para la detección y clasificación de las piezas, y los apriltags para la estimación la pose y la transformación al sistema de referencia de la base del robot. A continuación, se muestran las tablas de resultados finales obtenidas.

Tabla 4.1 Resultados del sistema 1 por iteraciones completas

Resultados SISTEMA 1 por iteraciones completas			
Acciones	Iteraciones	Aciertos	(%)
Detectar	90	82	91,11
Coger	90	80	88,89
Dejar 15%	30	24	80,00
Dejar 10%	30	19	63,33
Dejar 5%	30	10	33,33

Tabla 4.2 Resultados del sistema 1 por éxito concreto

Resultados SISTEMA 1 por éxito concreto			
Acciones	Iteraciones	Aciertos	(%)
Detectar	90	82	91,11
Coger	82	80	97,56
Dejar 15%	26	24	92,31
Dejar 10%	26	19	73,08
Dejar 5%	28	10	35,71

Como se puede observar en las tablas, el sistema de detección de la red neuronal tiene una eficacia del 91,11%. En cuanto a coger las piezas, en la Tabla 4.1 no alcanza en 90%, pero en la segunda tabla tiene un 97,56% de acierto. Esto se debe a que el éxito de la etapa individual es muy alto, pero en el sistema completo, un fallo en la detección de la pieza produce el fallo cuando se intenta coger. Dejar la pieza en la posición correcta depende de la tolerancia. Como es normal, a medida que disminuye la tolerancia disminuye el número de aciertos. Se puede observar cómo se va arrastrando el error durante el sistema. En la Tabla 4.1, con una tolerancia del 15%, el sistema tiene una fiabilidad del 80%, en cambio, si analizamos el éxito concreto de la etapa de dejar, con la misma tolerancia se obtiene una cantidad de aciertos superior al 90%.

4.2.2. SISTEMA 3: RNN | V.A. CLÁSICA | POINTCLOUD | APRILTAG-REF

En este sistema se ha utilizado la misma red neuronal que en el primer sistema para la detección y clasificación de los objetos, los métodos tradicionales para la adquisición de centro y vértices, y la nube de puntos para adquirir el punto y pose 3d. Posteriormente la transformación del sistema de referencia de la cámara al del robot se hace como en el sistema 1, con el apriltag de referencia. Los resultados obtenidos son los siguientes.

Tabla 4.3 Resultados del sistema 3 por iteraciones completas

Resultados SISTEMA 3 por iteraciones completas			
Acciones	Iteraciones	Aciertos	(%)
Detectar	90	85	94,44
Coger	90	69	76,67
Dejar 15%	30	21	70,00
Dejar 10%	30	17	56,67
Dejar 5%	30	7	23,33

Tabla 4.4 Resultados del sistema 3 por éxito concreto

Resultados SISTEMA 3 por éxito concreto			
Acciones	Iteraciones	Aciertos	(%)
Detectar	90	85	94,44
Coger	85	69	81,18
Dejar 15%	25	21	84,00
Dejar 10%	25	17	68,00
Dejar 5%	18	7	38,89

La detección es muy buena, alcanzando casi un 95% de aciertos. A la hora de coger la pieza, ya no hay tanta eficacia como en el sistema 1, disminuyendo la cifra de aciertos hasta un 81% en la etapa concreta. Esto se debe a que, el posicionamiento 3d del punto central de la pieza, utilizando la nube de puntos, es menos preciso que con apriltags. Como no podía ser de otra manera, al tener un sistema menos preciso, a la hora de dejar la pieza los aciertos fueron menos. Se alcanza un 84% con una tolerancia del 15%, un 68% con una tolerancia de 10%, y una escasa tasa de acierto de un 38% con la tolerancia más baja, 5%. Estos datos están reflejados en la tabla de éxito concreto, para así evaluar la precisión de cada etapa independientemente. En el siguiente apartado se hace una comparación de sistemas, donde la tabla de iteraciones completas toma mayor relevancia.

4.2.3. Comparación de sistemas

Comparando los resultados de los sistemas 1 y 3 se puede observar que el sistema 1 realiza mejor la labor completa, a pesar de que el sistema 3 tenga mejor valor en términos de detección e identificación, aunque tengan la misma red neuronal. Esto se debe a que en el sistema 1, las piezas tienen el apriltag en la cara superior, el cual confunde la red haciéndola menos precisa en términos de clasificación. En cuanto a coger la pieza, la combinación de los métodos tradicionales y la nube de puntos forman un sistema menos preciso que el de los apriltags, el cual está específicamente diseñado para realizar esta labor. Al ser menos preciso, esto se ve reflejado en la colocación de piezas, disminuyendo más de 10% de precisión en todas las tolerancias.

4.3. Discusiones

El presente proyecto ha agrupado y sincronizado componentes con el fin de solucionar propuesto al inicio, clasificar y colocar objetos. Los sistemas creados agrupan los componentes, y los utilizan para maximizar los objetivos.

Es importante el error total del sistema, se podría considerar elevado, un 20% de fallos en su versión con máxima tolerancia, pero hay que tener en cuenta que se pasa por una gran cantidad de componentes y métodos que van acumulando error.

Es posible reducir el error, empezando por el pegado de los apriltags, que se ha realizado 'a ojo'. Se podría continuar con la calibración de la cámara, importante para la detección de apriltags y para la nube de puntos. Se ha utilizado la calibración que viene hecha de fábrica, pero se podría probar una hecha en el laboratorio para ver cómo afecta al funcionamiento del sistema. Un error en la calibración altera todos los métodos de visión artificial. La resolución es crucial porque datos de mayor resolución son más precisos, lo que mejora los resultados de los métodos de detección y nube de puntos. Esto también afecta las matrices de transformación entre sistemas de referencia. En este proyecto, no se aumentó la resolución porque la cámara utilizada no lo permitía. El error se va arrastrando de uno a otro hasta llegar al sistema de la base del robot, que es el principal.

Como último ajuste, la posición exacta de los agujeros, el punto al que el robot se mueve para intentar meter la pieza en el agujero, y la posición del apriltag de referencia se ha calculado moviendo el robot a esos puntos, por lo que también se han ajustado manualmente. Todos estos ajustes manuales introducen errores en la cadena completa del sistema que se van a cumulando y dificultando la realización correcta de la tarea. Algunos de ellos pueden sustituirse por metodologías de calibración más sofisticadas que se proponen en los trabajos futuros.

Por último, habría que considerar el error de posicionamiento del robot, aunque vistos los resultados de nuestro sistema, pueden considerarse despreciables dado que son de un orden de magnitud inferior.

El presupuesto del proyecto se encuentra en el Anexo A. El enlace al video de YouTube mostrando el funcionamiento de los diferentes sistemas está en Anexo B.2.

4.4. DIFICULTADES

Tratándose de un proyecto abierto, han surgido dificultades en muchos aspectos, que se van a ir describiendo en orden cronológico.

La cámara utilizada tiene una característica particular, y es que permite cargar redes neuronales en el propio sistema de procesamiento de datos de la cámara, de manera que, en vez de entregarte una imagen, te entrega el resultado del su procesamiento en esa red. Sin embargo, la documentación del fabricante no aclara cómo puede adaptarse una red no creada por ellos para cargarla en la cámara. Esto hizo

que, tras muchos intentos, se optara por utilizar la cámara únicamente para la adquisición de imágenes RGB y la nube de puntos, y procesar la imagen con los métodos de detección elegidos en el ordenador.

El método de detección más complicado fue con la aplicación redes neuronales, por una parte, por la poca experiencia previa y, por otra parte, por las limitaciones computacionales que tenían los dispositivos utilizados. El etiquetado de imágenes también fue un factor limitante en el entrenamiento y un causante de los problemas de detección.

Capítulo 5. CONCLUSIONES

En este capítulo se recogen las conclusiones sobre el resultado final del proyecto y las áreas del proyecto actual para mejorar en el futuro.

5.1. CONCLUSIONES

En este Trabajo de Fin de Grado se ha abordado el desafío de implementar un sistema de *pick and place* utilizando el robot colaborativo UR3e, con el objetivo de detectar objetos y estimar su posición en el espacio tridimensional. Se han explorado y comparado diferentes enfoques para la detección de objetos y la estimación de su pose, incluyendo métodos tradicionales de visión artificial, técnicas basadas en aprendizaje profundo y enfoques de visión 3D.

Se han utilizado técnicas como la detección de AprilTags, redes neuronales convolucionales para la detección de objetos, métodos tradicionales como detección de contornos, así como la utilización de nubes de puntos para la adquisición de la pose en 3D de las piezas. Cada método ha sido implementado como un nodo distinto y pueden ser aplicados en el sistema general completo.

Los resultados experimentales muestran que cada técnica tiene sus ventajas y limitaciones. Por ejemplo, los métodos tradicionales de visión artificial son rápidos y pueden funcionar bien en condiciones controladas, pero son muy sensibles a cambios

en la iluminación y la textura de los objetos. Por otro lado, las redes neuronales convolucionales ofrecen una mayor capacidad de generalización y pueden aprender características más complejas de los objetos, pero requieren un gran esfuerzo de entrenamiento y pueden ser más lentas.

La utilización de nubes de puntos para la estimación de la pose en 3D ha demostrado ser efectiva para obtener información tridimensional precisa de los objetos, aunque reduce significativamente los FPS de la cámara.

En resumen, el proyecto proporciona una visión integral de diferentes enfoques para la detección de objetos y la estimación de su pose en un escenario *pick and place* con el robot. Los resultados y las conclusiones obtenidas pueden servir como punto de partida para futuros proyectos, definiendo sus posibilidades en el último apartado.

5.2. PERSPECTIVAS FUTURAS

El proyecto puede ser utilizado como punto de partida para el desarrollo de mejores sistemas de detección de objetos más complejos. Puede mejorarse afinando los parámetros o ampliando las funcionalidades del robot, como puede ser la libre elección de movimientos, la interrupción de un movimiento, la creación de trayectorias, el giro infinito de la pinza para labores de atornillado, etc.

Sobre la base proporcionada es posible el entrenamiento de modelos tanto *object detection* como *pose estimation* y su aplicación en el proyecto con el fin de lograr mejores resultados. También existe la posibilidad de mejorar o añadir sistemas de visión clásica más sofisticados y comprobar su funcionamiento y rendimiento en el proyecto.

Una clara mejora que se puede hacer es el cálculo de la matriz de transformación de la cámara a la base del robot, para no depender del AprilTag de referencia, aportando mayor flexibilidad al sistema. Para esto hay que calibrar la cámara respecto de la base del robot.

BIBLIOGRAFÍA

[1] J Edward Colgate, W. W. (1996). Cobots: Robots for collaboration with human operators. *ASME international mechanical engineering congress and exposition*, (págs. 433-439).

[2] IEB School. (s.f.). *¿Qué es un cobot? Robot colaborativo y tecnología*. Consultado en junio de 2024, de https://www.iebschool.com/blog/que-es-cobot-robot-colaborativo-tecnologia.

[3] Szeliski, R. (2010). Computer Vision, Algorithms and Applications. Springer.

[4] Universal Robots. (s.f.). *Universal Robots*. Consultado en junio de 2024, de https://www.universal-robots.com.

[5] Zimmer Group. (s.f.). *Zimmer Group*. Consultado en junio de 2024, de https://www.zimmer-group.com.

[6] Luxonis. (s.f.). *Luxonis Documentation*. Consultado en junio de 2024, de https://docs.luxonis.com.

[7] University of Michigan. (s.f.). *Apriltag*. Consultado en junio de 2024, de https://april.eecs.umich.edu/software/apriltag.html.

[8] Ponce, P. (2011). Inteligencia artificial con aplicaciones a la ingenieria. Marcombo.

[9] Ultralytics. (s.f.). *YOLOv8 Documentation*. Consultado en junio de 2024, de https://docs.ultralytics.com.

[10] Roboflow. (s.f.). *Roboflow*. Consultado en junio de 2024, de https://roboflow.com.

[11] Google. (s.f.). *Google Colab*. Consultado en junio de 2024, de https://colab.research.google.com.

[12] Stephens, C. H. (1988). A Combined Corner and Edge Detector. *Alvey Vision Conference*.

[13] Universal Robots. (s.f.). *Centro de Descargas - Universal Robots*. Consultado en mayo de 2024, de https://www.universal-robots.com/es/centro-de-descargas/#/e-series/ur3e.

[14] Alexander Mayer, V. C.-V. (2022). *The Universal Robots Real-Time Data Exchange (RTDE) and LabVIEW*.

[15] Luxonis. (s.f.). *DepthAI: Embedded Machine Learning and Computer Vision API*. Consultado en junio de 2024, de https://luxonis.com.

[16] Luxonis. (s.f). *OAK-D: Stereo Camera with Edge AI.* Consultado en junio de 2024, de https://luxonis.com

[17] Open3D. (2024). *Open3D: Open-Source Library for 3D Data Processing*. Consultado en junio de 2024, de https://www.open3d.org

[18] Rosebrook, A. (s.f.). *Practical Python and OpenCV + Case Studies: An Introductory, Example Driven Guide to Image Processing and Computer Vision*. PyImageSearch.

[19] Edwin Olson, N. M. (2011). AprilTag: A robust and flexible visual fiducial system. *International Conference on Computer Vision Systems*.

[20] Alex Krizhevsky, I. S. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*.

[21] Canny, J. (1986). A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Anexo A. PRESUPUESTO

Para el presupuesto del presente proyecto se va a tener en cuenta los costes directos e indirectos, desglosados a continuación:

Costes indirectos

Tabla A.1 Costes indirectos

Costes indirectos	€/mes	meses	Total €
electricidad e internet	45	6	270

Costes directos

Se tienen en cuenta los costes en mano de obra y materiales.

- Estudiante: Siendo el salario medio de un ingeniero junior en España de 25.001€ anuales, el precio por hora es de 12.82 €/h.
- Tutor: Siendo el salario medio de un profesor universitario en España de 33.100€, el precio por hora es de 22.39 €/h.

Tabla A.2 Horas dedicadas por la mano de obra

Mano de obra	horas/dia	dias/semana	semanas	horas totales
Estudiante	4	5	20	400
Tutor	1,5	1	20	30

Tabla A.3 Costes de la mano de obra

Mano de obra	horas	€/hora	TOTAL €
Estudiante	400	12,82	5128
Tutor	30	22,39	671,7
Total			**5799,7**

Tabla A.4 Coste de los materiales

Materiales	Coste €
Robot UR3e	18000
Pinza HRC-03	195
Ordenador	1050
Cámara OAK-D-Lite	180
Cable USB-C	12
Piezas y soporte cámara	4
Maderas	2
Tornillos	1,5
Total	**19444,5**

Tabla A.5 Costes directos

Costes directos	Importe €
Mano de obra	5799,7
Materiales	19444,5
Total	**25244,2**

Costes totales

Tabla A.6 Costes totales

Costes	Importe
Costes directos	25244,2
Costes Indirectos	270
Total	**25514,2**

Anexo B. ENLACES

B.1. ENLACE AL REPOSITORIO DE GITHUB

https://github.com/dtertre59/TFG

B.2. ENLACE AL VIDEO DE YOUTUBE

https://youtu.be/vQbSdDP40Zg

Anexo C. TABLAS

Las siguientes tablas muestran los resultados obtenidos en las pruebas de funcionamiento de los sistemas 1 y 3 al completo. Se evalúa el acierto o fallo de las diferentes etapas del sistema, que son detectar, coger y dejar. Se han realizado un total de 10 iteraciones por cada pieza y por cada tolerancia, los que da un total de 90 iteraciones por sistema. El acierto de la etapa se puntúa con un 1 y el fallo con un 0. El fallo de una etapa produce automáticamente el fallo en las etapas que la suceden.

Tabla A.7 Resultados del sistema 1

SISTEMA 1													
Piezas	CUADRADO				CÍRCULO				HEXÁGONO				
	Squar	Detecta	Coge	Dejar 15%	Circul	Detecta	Coge	Dejar 15%	Hexágon	Detecta	Coge	Dejar 15%	
tolerancia 15%	1	1	1	1	1	1	1	1	1	1	1	1	
	2	0	0	0	2	1	1	1	2	1	1	1	
	3	1	1	1	3	1	1	1	3	1	1	1	
	4	1	1	1	4	1	1	1	4	0	0	0	
	5	1	1	1	5	1	1	1	5	0	0	0	
	6	1	1	0	6	1	1	1	6	1	1	1	
	7	1	1	1	7	1	1	1	7	1	1	1	
	8	1	1	1	8	1	1	1	8	0	0	0	
	9	1	1	1	9	1	1	1	9	1	1	0	
	10	1	1	1	10	1	1	1	10	1	1	1	
	TOTAL	9	9	8	TOTAL	10	10	10	TOTAL	7	7	6	
	Squar	Detecta	Coge	Dejar 10%	Circul	Detecta	Coge	Dejar 10%	Hexágon	Detecta	Coge	Dejar 10%	
tolerancia 10%	1	1	1	1	1	1	1	0	1	1	1	1	
	2	1	1	0	2	1	1	1	2	1	1	1	
	3	1	1	1	3	1	1	1	3	1	1	0	
	4	1	1	1	4	1	1	0	4	1	1	1	
	5	1	1	1	5	1	1	1	5	1	1	0	
	6	1	1	1	6	1	1	1	6	0	0	0	
	7	1	0	0	7	1	1	1	7	1	1	0	
	8	1	0	0	8	1	1	0	8	1	1	1	
	9	1	1	1	9	1	1	1	9	0	0	0	
	10	1	1	1	10	1	1	1	10	1	1	1	
	TOTAL	10	8	7	TOTAL	10	10	7	TOTAL	8	8	5	
	Squar	Detecta	Coge	Dejar 5%	Circul	Detecta	Coge	Dejar 5%	Hexágon	Detecta	Coge	Dejar 5%	
tolerancia 5%	1	1	1	0	1	1	1	0	1	1	1	0	
	2	1	1	0	2	1	1	0	2	1	1	1	
	3	1	1	0	3	1	1	1	3	1	1	0	
	4	1	1	1	4	1	1	0	4	1	1	0	
	5	1	1	0	5	1	1	0	5	1	1	1	
	6	1	1	1	6	1	1	0	6	1	1	0	
	7	1	1	1	7	1	1	0	7	0	0	0	
	8	1	1	1	8	1	1	1	8	1	1	0	
	9	1	1	1	9	1	1	1	9	1	1	0	
	10	0	0	0	10	1	1	0	10	1	1	0	
	TOTAL	9	9	5	TOTAL	10	10	3	TOTAL	9	9	2	

Tabla A.8 Resultados del sistema 3

SISTEMA 3												
Piezas	CUADRADO				CÍRCULO				HEXÁGONO			
	Squar	Detecta	Coge	Dejar 15%	Circul	Detecta	Coge	Dejar 15%	Hexágon	Detecta	Coge	Dejar 15%
tolerancia 15%	1	1	1	1	1	1	1	1	1	1	1	1
	2	1	1	1	2	1	1	0	2	1	1	1
	3	1	1	1	3	1	1	1	3	1	1	0
	4	1	1	1	4	1	1	1	4	1	1	1
	5	1	0	0	5	1	1	1	5	1	1	0
	6	1	1	1	6	1	1	1	6	1	1	1
	7	1	0	0	7	1	1	1	7	1	1	0
	8	1	1	1	8	1	0	0	8	1	1	1
	9	1	0	0	9	1	1	1	9	0	0	0
	10	1	1	1	10	1	1	1	10	1	1	1
	TOTAL	10	7	7	TOTAL	10	9	8	TOTAL	9	9	6
	Squar	Detecta	Coge	Dejar 10%	Círcul	Detecta	Coge	Dejar 10%	Hexágon	Detecta	Coge	Dejar 10%
tolerancia 10%	1	1	1	0	1	1	1	0	1	1	1	0
	2	1	1	1	2	1	1	0	2	1	1	1
	3	1	1	1	3	1	1	1	3	1	1	1
	4	1	1	1	4	1	1	0	4	1	1	1
	5	1	0	0	5	1	1	1	5	1	1	1
	6	1	0	0	6	1	1	0	6	0	0	0
	7	1	1	0	7	1	1	1	7	1	1	1
	8	1	1	1	8	1	1	1	8	1	1	0
	9	1	1	1	9	1	1	1	9	1	1	1
	10	1	1	1	10	1	1	0	10	0	0	0
	TOTAL	10	8	6	TOTAL	10	10	5	TOTAL	8	8	6
	Squar	Detecta	Coge	Dejar 5%	Círcul	Detecta	Coge	Dejar 5%	Hexágon	Detecta	Coge	Dejar 5%
tolerancia 5%	1	1	0	0	1	1	1	1	1	1	1	1
	2	1	0	0	2	1	1	1	2	1	1	0
	3	1	1	0	3	1	1	0	3	1	1	1
	4	1	1	0	4	1	1	1	4	1	1	0
	5	1	0	0	5	1	1	0	5	0	0	0
	6	1	0	0	6	1	0	0	6	1	1	0
	7	1	1	1	7	0	0	0	7	1	0	0
	8	1	0	0	8	1	1	0	8	1	1	0
	9	1	0	0	9	1	1	0	9	1	0	0
	10	1	1	0	10	1	0	0	10	1	1	1
	TOTAL	10	4	1	TOTAL	9	7	3	TOTAL	9	7	3