



Centro Universitario de la Defensa en la Escuela Naval Militar

TRABAJO FIN DE GRADO

*Desarrollo de un sistema de detección de embarcaciones
mediante visión artificial con YOLO y Unity*

Grado en Ingeniería Mecánica

ALUMNO: José Manuel Ros Rial

DIRECTOR: Miguel Rodelgo Lacruz

CURSO ACADÉMICO: 2025-2026

Universida_{de}Vigo



Centro Universitario de la Defensa en la Escuela Naval Militar

TRABAJO FIN DE GRADO

*Desarrollo de un sistema de detección de embarcaciones
mediante visión artificial con YOLO y Unity*

Grado en Ingeniería Mecánica
Intensificación en Tecnología Naval
Cuerpo General

UniversidadeVigo

RESUMEN

Este Trabajo de Fin de Grado tiene como objetivo el desarrollo e implementación de una aplicación móvil Android capaz de detectar embarcaciones en tiempo real mediante el uso de técnicas de visión artificial. La primera versión se ha desarrollado utilizando el motor Unity y un modelo YOLO (*You Only Look Once*) pre-entrenado con un *dataset* genérico (COCO) para la identificación de barcos y, posteriormente, se ha entrenado un modelo propio con un *dataset* especializado (*SeaShips*) para mejorar la clasificación y el rendimiento. El sistema procesa las imágenes capturadas por la cámara del dispositivo móvil y es capaz de detectar las embarcaciones presentes en la escena, generando un marcador que permite seguir su localización dentro de la imagen. De forma complementaria, se ha realizado la evaluación del rendimiento del sistema mediante el uso de conjuntos de datos específicos, con el fin de analizar su comportamiento en distintos escenarios y condiciones de operación.

Como aplicación independiente, la solución propuesta puede resultar útil en tareas de conteo de embarcaciones, registro visual del tráfico marítimo y como apoyo a labores de vigilancia y supervisión en entornos portuarios o costeros. Además, el sistema se concibe con un módulo integrable en la línea de Trabajos Fin de Grado del Centro Universitario de la Defensa, centrados en la representación de datos AIS mediante realidad aumentada. La combinación de la detección visual con la información AIS permitiría corregir la notable desviación observada de los sensores de los visores (IMU, GPS y, especialmente, magnetómetro) en el posicionamiento de los marcadores virtuales sobre la escena real.

PALABRAS CLAVE

YOLO, Unity, *Deep Learning*, Embarcaciones, Inteligencia Artificial

AGRADECIMIENTOS

En primer lugar, me gustaría profundamente agradecer a mi familia y mi pareja ya que, sin su cariño, su apoyo incondicional y los consejos que me han brindado a lo largo de los años no sería capaz de escribir estas palabras. Son un pilar fundamental en mi vida, mi razón de ser y son los que me han empujado a completar mis cinco años de formación en la Escuela Naval Militar.

Por otra parte, me gustaría agradecer a compañeros y hermanos de la promoción 426-156, Alexander, Adrián y Pablo con los cuales he vivido tantas anécdotas dentro de la escuela. Agradezco también a mi tutor, D. Miguel Rodelgo, por la ayuda y guía durante el desarrollo del presente Trabajo de Fin de Grado, resolviendo mis dudas y supervisando cada avance del trabajo.

CONTENIDO

Contenido	1
Índice de Figuras	3
Índice de Tablas.....	5
1 Introducción y objetivos	7
1.1 Contexto	7
1.2 Motivación del trabajo e identificación del problema.	8
1.3 Objetivos	9
1.4 Estructura	10
2 Estado del arte	11
2.1 Inteligencia Artificial	11
2.1.1 Origen de la Inteligencia Artificial	11
2.2 Redes neuronales.....	12
2.2.1 Origen e historia	12
2.2.2 Tipos de redes	13
2.3 Visión Artificial	14
2.4 Detección de objetos con redes neuronales profundas.....	17
2.4.1 Faster R-CNN	17
2.4.2 RetinaNet	17
2.4.3 SSD	17
2.4.4 YOLO	17
2.5 Detección de objetos con Yolo	18
2.5.1 Funcionamiento de YOLO	18
2.5.2 Versiones y evolución histórica.....	19
2.6 Conjunto de datos.....	21
2.6.1 COCO	21
2.6.2 Marvel.....	23
2.6.3 SeaShips.....	23
2.6.4 Datasets de imágenes aéreas	24
2.7 Entorno de trabajo	25
2.7.1 Unreal Engine	25
2.7.2 Android Studio.....	25
2.7.3 Unity	25
3 Desarrollo del TFG.....	27
3.1 Dispositivos empleados.....	27

3.1.1 Ordenador	27
3.1.2 Dispositivo móvil.....	28
3.2 Configuración general de Unity	28
3.3 Funcionalidades de la aplicación	30
3.4 Desarrollo de la aplicación.....	32
3.4.1 Escena principal.....	32
3.4.2 Menú de opciones	37
3.4.3 Escena inicial	39
3.5 Entrenamiento con dataset específico.	40
4 Validación y pruebas	43
4.1 Resultados de los modelos (COCO vs propio)	43
4.1.1 Comparativa (conjunto de validación SeaShips)	43
4.2 Prueba de la aplicación en entorno real	46
5 Conclusiones y líneas futuras	49
5.1 Conclusiones y revisión de los objetivos	49
5.2 Líneas futuras	50
6 Bibliografía.....	51
Anexo I: Script yolodetector.cs	55
Anexo II: Script optionmenucontroller.cs	67
Anexo III: Script startscreencontroller.cs	69
Anexo IV: Script servidor gpu	73
Anexo V: Script detectorconfig.cv	79
Anexo VI: Script pruebas servidor	81
Anexo VII: Uso de inteligencia artificial	85
Anexo VIII: Implicaciones sociales, económicas, y ambientales	89
Anexo IX: Reflexiones éticas y sociales	91

ÍNDICE DE FIGURAS

Figura 1-1 Ejemplificación detección de barcos (Elaboración propia)	7
Figura 1-2 Ejemplificación del error de magnetómetro [6].....	9
Figura 2-1 Representación Prueba de Turing (Elaboración propia).....	11
Figura 2-2 Línea de evolución de la Inteligencia Artificial (Elaboración propia)	12
Figura 2-3 Esquema fases visión artificial (Elaboración propia)	14
Figura 2-4 Funcionamiento de sistema de detección de objetos (Elaboración propia).....	16
Figura 2-5 Esquema funcionamiento de YOLO (Elaboración propia)	19
Figura 2-6 Dataset COCO [45]	22
Figura 2-7 Dataset Marvel [47]	23
Figura 2-8 Dataset SeaShips [48]	24
Figura 2-9 Datasets aéreos [49].....	24
Figura 2-10 Logos de Unity, Unreal Engine y Android Studio [51],[52],[53]	25
Figura 3-1 Asus GL502VS (Elaboración propia).....	27
Figura 3-2 Xiaomi Redmi Note 11 (Elaboración propia).....	28
Figura 3-3 Entorno Unity Hub (Elaboración propia)	29
Figura 3-4 Escena general de Unity (Elaboración propia)	29
Figura 3-5 Menú Package Management (Elaboración propia)	30
Figura 3-6 Escena principal (Elaboración propia).....	32
Figura 3-7 Representación FadeOverlay (Elaboración propia).....	32
Figura 3-8 Representación WebcamScreen (Elaboración propia)	33
Figura 3-9 Representación TextBackground (Elaboración propia)	33
Figura 3-10 Representación Detection Panel (Elaboración propia).....	33
Figura 3-11 Representación Botón ajustes (Elaboración propia).....	34
Figura 3-12 Representación ZoomSlider (Elaboración propia)	34
Figura 3-13 Panel variables YoloDetector (Elaboración propia).....	34
Figura 3-14 Representación botones de cerrar y salida (Elaboración propia).....	37
Figura 3-15 Representación Sliders (Elaboración propia)	37
Figura 3-16 Escena menú (Elaboración propia).....	38
Figura 3-17 Panel variables DetectorOptionsMenu(Elaboración propia)	38
Figura 3-18 Escena de inicio (Elaboración propia).....	39
Figura 3-19 Audio Source Panel (Elaboración propia)	40
Figura 3-20 Comparativa de modelos (Elaboración propia)	42
Figura 4-1 Gráfica comparación (Elaboración propia)	43
Figura 4-2 Métricas de clasificación (Elaboración propia)	44

Figura 4-3 Comparativa detección N°1 (Elaboración propia).....	45
Figura 4-4 Comparativa detección N°2 (Elaboración propia).....	45
Figura 4-5 Comparativa detección N°3 (Elaboración propia).....	45
Figura 4-6 Comparativa detección N°4 (Elaboración propia).....	46
Figura 4-7 Captura “Saliendo de la dársena” (Elaboración propia).....	46
Figura 4-8“Entre puntas” (Elaboración propia)	47
Figura 4-9“Ría de Pontevedra” (Elaboración propia)	47

ÍNDICE DE TABLAS

Tabla 2-1 Tipos de redes neuronales[14], [15].....	13
Tabla 2-2 Tabla comparativa métodos clásicos frente a modernos[21]	15
Tabla 2-3 Comparativa versiones de YOLO (Extraído de Ultralytics)[31]	19
Tabla 3-1 Características Asus GL502VS.....	28
Tabla 3-2Características Xiaomi Readmi Note 11	28

1 INTRODUCCIÓN Y OBJETIVOS

El presente capítulo tiene como finalidad introducir los elementos que conforman este proyecto, centrándose en el ámbito temático de la detección de embarcaciones mediante visión artificial. En sus apartados se detalla el proceso y las razones que motivaron la elección del tema objeto de estudio, constituyendo así el comienzo para el desarrollo del presente Trabajo de Fin de Grado (TFG).

1.1 Contexto

El ámbito naval actual sumado a la inestabilidad del entorno geopolítico internacional exige que las Fuerzas Armadas, y en especial la Armada, intensifiquen sus actividades de vigilancia marítima en áreas de interés nacional, garantizando así una ventaja estratégica y la seguridad en la libre navegación de las aguas de ámbito nacional. En este marco, la Armada Española, dentro de su esquema de maniobra, tiene como objetivo operativo fundamental el mantener el control del mar en escenarios cada vez más saturados de información [1].

Para responder a este desafío, se vuelve un factor clave la integración de sensores avanzados, como son: Los sistemas de posicionamiento global (GPS), transpondedores del Sistema de Identificación Automática (AIS), radares de superficie, sonares y cámaras electroópticas e infrarrojas. Estos sensores producen un flujo de información que si se gestiona de manera correcta y eficiente que proporciona una imagen completa y dinámica del entorno marítimo. La fusión de estos datos, junto con el uso de sistemas de visión artificial basados en Inteligencia Artificial, permite una vigilancia más precisa y eficiente. En particular, los algoritmos de detección de objetos en tiempo real mediante visión artificial como YOLO (*You Only Look Once*) [2] emergen como herramientas disruptivas para procesar y analizar grandes volúmenes de información, optimizando la toma de decisiones y reforzando la capacidad de respuesta ante posibles amenazas o incidencias en el mar como se ve ejemplificado en la Figura 1-1.

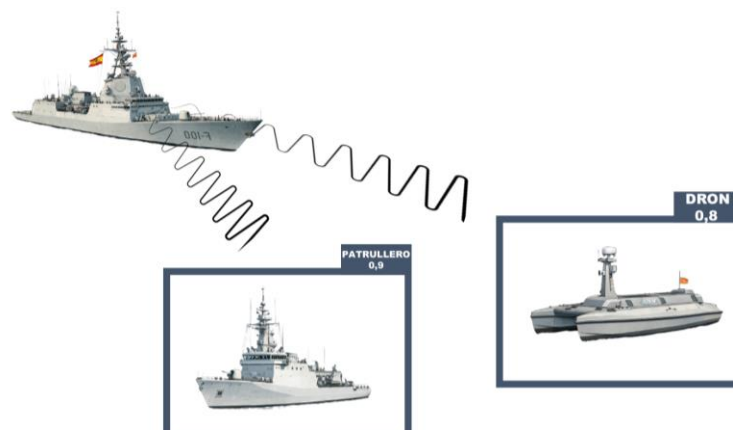


Figura 1-1 Ejemplificación detección de barcos (Elaboración propia)

Esta tecnología tiene la oportunidad de potenciar los sensores convencionales, que se encuentran en buques de vigilancia portuaria, permitiendo una identificación y clasificación de embarcaciones, reduciendo la dependencia del factor humano. La integración de modelos como YOLO en los sistemas de vigilancia existentes representa una oportunidad para automatizar el análisis del entorno y favorecer una adaptación a las nuevas tecnologías de reconocimiento del entorno marítimo.

1.2 Motivación del trabajo e identificación del problema.

La capacidad de vigilancia y el conocimiento del entorno naval constituyen uno de los pilares del poder naval moderno. Las armadas deben garantizar la seguridad marítima y proyectar una disuasión frente a posibles amenazas. En el caso de la Armada española, los sistemas de observación convencionales y la vigilancia por parte de personal cualificado han desarrollado un papel clave en la protección de los intereses nacionales. No obstante, la dependencia exclusiva del factor humano para la vigilancia continua implica limitaciones fisiológicas, como la fatiga visual y la saturación mental, así como elevados costes asociados a la necesidad de mantener dotaciones numerosas en estado alerta constante.

De forma paralela, el entorno estratégico marítimo ha experimentado una evolución significativa debido a la innovación tecnológica y la diversificación de las amenazas. El incremento exponencial del tráfico marítimo, sumado a la necesidad de proteger infraestructuras críticas y la aparición de escenarios de conflicto asimétrico los cuales se caracterizan por ser amenazas rápidas y de baja firma radar, como por ejemplo drones o embarcaciones ligeras han generado nuevas exigencias operativas. Estas demandas de reconocimiento y procesamiento de información visual no siempre pueden ser cubiertos de manera eficaz y eficiente por operadores saturados de datos. En este contexto, surge la necesidad de complementar los sensores tradicionales con sistemas de Inteligencia Artificial (IA) capaces de procesar imágenes en tiempo real, asumiendo tareas de detección y clasificación sin los factores fisiológicos intrínsecos al ser humano.

Ante este escenario, la Armada Española se enfrenta al reto de mantener un ritmo de desarrollo de este tipo de tecnologías ya que la capacidad para analizar y procesar el entorno marítimo se ha convertido en un pilar fundamental para garantizar la seguridad y la operatividad en el ámbito naval actual. El desarrollo e implementación de un sistema de detección de objetos mediante visión artificial representa un desafío de gran interés. Este tipo de proyectos requiere la integración de conocimientos avanzados en diferentes disciplinas, como el entrenamiento de redes neuronales convolucionales, programación y el desarrollo de aplicaciones.

Factores como la variabilidad de la iluminación, la inestabilidad provocada por el oleaje, la interferencia meteorológica y la necesidad de procesamiento en tiempo real son retos inherentes que ponen a prueba la robustez de los sistemas inteligentes [3].

Además de sus usos como aplicación independiente, el trabajo se concibe principalmente como un módulo capaz de corregir la principal limitación detectada en los trabajos fin de grado del AN Ángel Gamboa Ramos Izquierdo[4], y el AN Pablo Díaz García[5], en los que se desarrolló una aplicación móvil de realidad aumentada en Unity que superpone, sobre la imagen captada por la cámara, marcadores virtuales en las posiciones geográficas de las embarcaciones recibidas mediante AIS (*Automatic Identification System*). Dichos marcadores se sitúan sin analizar visualmente la imagen, en base a los sensores de posicionamiento del dispositivo -IMU, GPS y magnetómetro-, por lo que sus errores, en especial de este último, provocan una desviación muy aparente de los contactos como se aprecia en la Figura 1-2 [6]. La integración de este trabajo en la aplicación previa permitiría asignar cada marcador a la embarcación más cercana detectada visualmente, corrigiendo las desviaciones y manteniendo el posicionamiento adecuado del marcador entre recepciones AIS. Este objetivo condiciona la elección de Unity y Android como plataforma de desarrollo para este proyecto.



Figura 1-2 Ejemplificación del error de magnetómetro [6]

1.3 Objetivos

El objetivo principal de este proyecto es el diseño, implementación y evaluación de un sistema de detección y seguimiento automático de embarcaciones basado en visión artificial, empleando Unity y Android como plataforma de desarrollo.

Para alcanzar este propósito, se establecen una serie de objetivos secundarios que estructuran el desarrollo del sistema:

1. Estudio del estado del arte y selección del modelo de detección de objetos.
2. Desarrollo de la aplicación con un modelo preeentrenado.
3. Entrenamiento y optimización del modelo mediante *datasets* especializados.
4. Desarrollo de la interfaz de visualización.
5. Pruebas en entornos reales

1.4 Estructura

Este trabajo se estructura en 6 capítulos que permiten comprender de forma ordenada y coherente cada una de las etapas del proyecto.

En el primer capítulo se presenta el contexto, la motivación de este proyecto. Además, se definen de manera clara los objetivos principales que debe alcanzar el sistema de visión artificial propuesto.

Prosiguiendo con el segundo capítulo se expone el estado del arte, abordando desde la inteligencia artificial centrado en la visión artificial hasta las tecnologías de detección. También se incluye un análisis de los entornos de trabajo relevantes para el desarrollo.

Dentro del tercer capítulo se detalla desarrollo de la aplicación de detección de embarcaciones, explicando los recursos empleados, además de su estructura y código. Asimismo, se describe paso a paso el proceso de entrenamiento del modelo para garantizar su correcta implementación.

En el cuarto capítulo se exponen los resultados y la validación del sistema mediante una evaluación cuantitativa basada en métricas de desempeño. De igual forma, se presenta unas pruebas experimentales realizadas en entorno real para comprobar la efectividad de la herramienta.

Continuando con el quinto capítulo se recogen las conclusiones, dando una discusión final sobre la eficacia y el impacto de la solución desarrollada. Para terminar, se proponen diversas líneas de trabajo futuro que podrían ampliar, optimizar o dar continuidad al sistema en próximos proyectos.

En última instancia dentro del sexto capítulo se recopilan todas las referencias bibliográficas. Junto a esto, se añaden los anexos e información adicional que sirve de apoyo para comprender mejor los aspectos más específicos del trabajo.

2 ESTADO DEL ARTE

En el presente capítulo se expondrán los fundamentos de la Inteligencia Artificial, centrando el análisis sobre la Visión Artificial como rama especializada en la interpretación y el procesamiento de información visual. Posteriormente, se expondrán las tecnologías y los recursos empleados en este tipo de proyectos, como los modelos de detección de objetos como YOLO, y los conjuntos de datos, que se emplean para el entrenamiento y la validación de los modelos como *SeaShips*. Asimismo, se abordarán motores gráficos como herramienta de apoyo al desarrollo, destacando su uso como entorno principal para la implementación, integración y visualización.

2.1 Inteligencia Artificial

La inteligencia artificial es una tecnología multidisciplinar e innovadora la cual ha revolucionado todos los ámbitos de la sociedad actual, pero cuyo nacimiento se remonta unas cuantas décadas.

2.1.1 Origen de la Inteligencia Artificial

El origen de la Inteligencia Artificial suele situarse en la segunda mitad del siglo XX, cuando comenzó a definirse como un ámbito de investigación autónomo. En 1950, Alan Turing formuló de manera explícita la duda de que una máquina pudiera mostrar comportamiento inteligente y expuso un criterio para su evaluación, posteriormente conocido como Test de Turing.

Esta prueba que se puede observar en la Figura 2-1, consistía en plantear una conversación escrita en el que un evaluador interactúa con dos emisores ocultos (un humano y una máquina), de forma que si no consigue distinguir de manera coherente cuál de ellos es la máquina a partir de sus respuestas, se considera que esta ha superado la prueba [7].

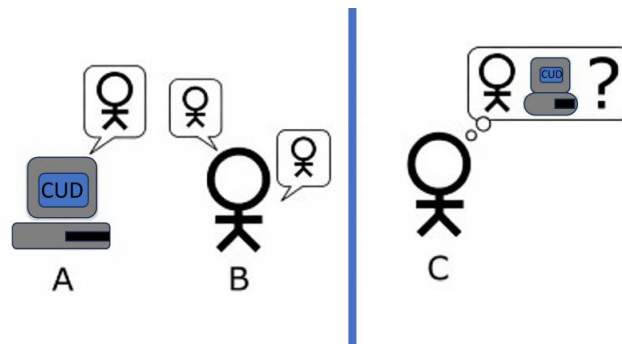


Figura 2-1 Representación Prueba de Turing (Elaboración propia)

Posteriormente, en 1956, la reunión de Dartmouth [8] consolidó el nacimiento formal de la disciplina, al acuñar el término inteligencia artificial y establecer un programa de investigación orientado a modelar computacionalmente capacidades como el razonamiento y el aprendizaje. En sus primeras etapas, el campo se estructuró principalmente en torno a enfoques simbólicos, basados en lógica y sistemas de reglas, como vía para representar conocimiento y resolver problemas.

Después en la década de 1980, los modelos de IA llamados sistemas expertos [9] fueron incorporados de forma generalizada por empresas a nivel mundial, y el conocimiento pasó a ocupar un lugar central en la investigación en cuanto a IA. A la vez, los gobiernos incrementaron de manera notable la financiación. Como se ve en la cronología de la Figura 2-2, La IA se sigue desarrollando hasta la actualidad como, por ejemplo, con la invención de ChatGPT.

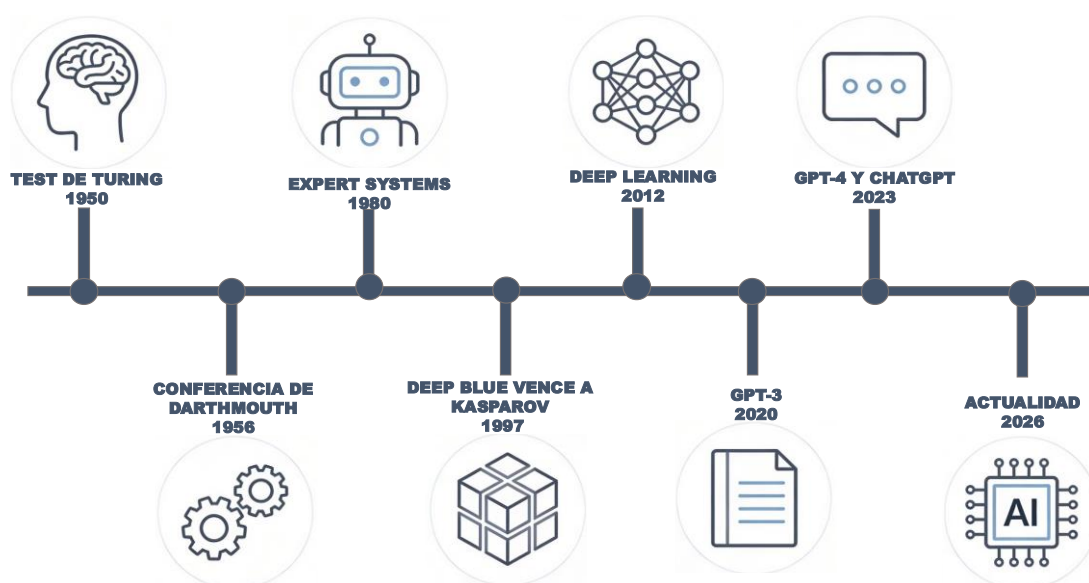


Figura 2-2 Línea de evolución de la Inteligencia Artificial (Elaboración propia)

2.2 Redes neuronales

Las redes neuronales artificiales constituyen uno de los pilares fundamentales de la inteligencia artificial moderna. Su desarrollo ha sido un proceso evolutivo que abarca más de ocho décadas, desde los primeros modelos matemáticos inspirados en el cerebro humano hasta las complejas arquitecturas de aprendizaje profundo que impulsan las aplicaciones de IA actuales. Este apartado presenta un recorrido histórico por los hitos más significativos en el desarrollo de las redes neuronales, analizando su origen conceptual, los períodos de auge y declive, y los avances recientes que han revolucionado el campo de la inteligencia artificial.

2.2.1 Origen e historia

El origen estricto de lo que conocemos hoy en día como redes neuronales artificiales data de 1943, seis años antes de que el conocido Donald Hebb formara las bases de la Teoría de las Redes Neuronales. Cuando Sturgis y Pitts publicaron su artículo (*A Logical Calculus of the Ideas Immanent in Nervous Activity*) [10] sentaron las bases del primer modelo de una neurona artificial, denominado como “la neurona de McCulloch-Pitts”, que revolucionaría la forma de imaginar el procesamiento de información. La neurona artificial se inspiró en el funcionamiento de las biológicas y sus conexiones. A diferencia de los modelos computacionales tradicionales, este modelo funcionaba mediante un sistema de lógica fundamentado en matemáticas y algoritmos. Además, demostró ser capaz de usar correctamente las funciones booleanas básicas: AND, OR y NOT, estableciendo que redes formadas por estas neuronas podían, en teoría, calcular cualquier función. Este descubrimiento fundamental asentó las bases de dos

líneas de investigación: una focalizada en los procesos de la mente humana y otra centrada en aplicaciones prácticas de inteligencia artificial.

Unos años después en 1958 Rosenblatt introduce el concepto de perceptrón, un modelo capaz de aprender patrones a partir de ejemplos. Se fundamentaba en clasificar entradas en dos tipos mediante un proceso de aprendizaje vigilado. Durante el entrenamiento del modelo, este ajustaba sus pesos mediante un procedimiento de corrección de errores por retropropagación. Este mecanismo permitía que el perceptrón aprendiera de sus errores y mejorara progresivamente su desempeño [11].

El verdadero punto de inflexión en las redes neuronales vino en 2010 con el incipiente desarrollo del *Deep Learning* [12]. Este concepto se caracteriza por el uso de redes neuronales agrupadas en múltiples capas ocultas, capaces de aprender representaciones jerárquicas de datos cada vez más abstractas. Esto ha permitido a las redes neuronales procesar enormes cantidades de datos no estructurados, como pueden ser imágenes, audio o texto, con una precisión nunca vista.

Los avances se deben a diversas causas como: el incremento exponencial del poder computacional, específicamente en el ámbito de las GPU, que ha permitido entrenar redes con un número elevado de parámetros; el acceso masivo de datos en la era digital que sirve como materia prima fundamental para el entrenamiento de estos modelos; y, por último, nuevos modelos de redes neuronales como pueden ser las redes neuronales convolucionales, las redes neuronales recurrentes y *Transformers*. Se han desarrollado aplicaciones prácticas revolucionarias como puede ser el reconocimiento facial, los asistentes virtuales, los traductores automáticos y sistemas de recomendación personalizados, exhibiendo la habilidad de la IA para entender y producir información [13].

2.2.2 Tipos de redes

Las redes neuronales artificiales se han diversificado en múltiples arquitecturas especializadas, cada una diseñada para resolver tipos específicos de problemas en inteligencia artificial (Tabla 2-1). Las CNN son las de mayor relevancia para este proyecto, ya que están especializadas en la extracción e identificación automática de patrones visuales y formas complejas directamente en imágenes [14].

Tipo	Estructura principal	Datos de entrada	Aplicaciones	Ventajas clave
Feedforward	Capas secuenciales sin bucles	Datos tabulares/estáticos	Clasificación, regresión	Simplicidad, entrenamiento rápido
CNN	Filtros + pooling + capas densas	Imágenes, grids espaciales	Visión artificial, detección objetos	Eficiencia espacial, pocos parámetros
RNN/LSTM/GRU	Bucles recurrentes con memoria	Secuencias temporales	Series temporales, NLP básico, señales	Memoria contextual secuencial
Transformers	Self-attention multi-cabeza	Texto, secuencias largas	Traducción, chatbots, visión multimodal	Paralelización, dependencias largas

Tabla 2-1 Tipos de redes neuronales[14], [15]

2.3 Visión Artificial

La Visión Artificial es una de las ramas de la Inteligencia Artificial[16] cuyo principal objetivo es proporcionar la capacidad de analizar, interpretar y extraer información relevante a partir de imágenes y vídeos. A diferencia del procesamiento de imágenes clásico[17], que suele estar pensado para la mejora de la calidad visual para, por ejemplo, la reducción de ruido en fotogramas o secuencias de vídeo, la Visión Artificial busca un enfoque más amplio, donde el objetivo es analizar el contenido de la escena, procesar los datos y generar información que pueda ser utilizada directamente en procesos de decisión automatizados. La Figura 2-3 Esquema fases visión artificial (Elaboración propia) representa un esquema de cada fase.

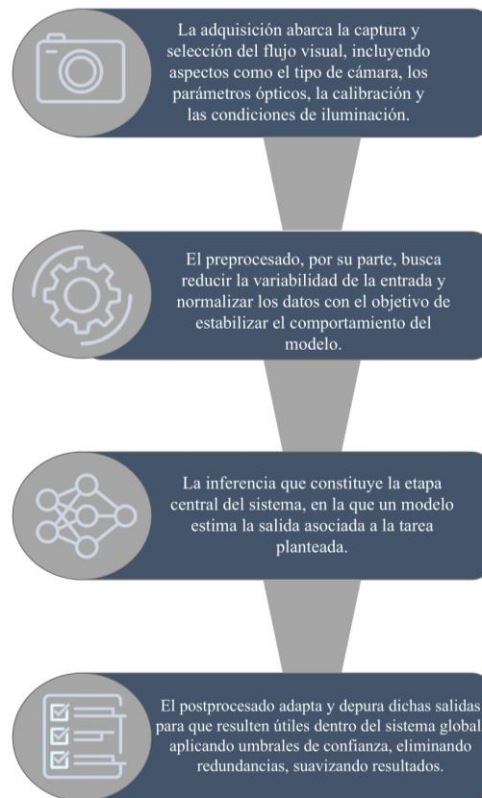


Figura 2-3 Esquema fases visión artificial (Elaboración propia)

En este sentido, no se trata únicamente de mejorar la imagen, sino de saber lo que aparece en ella y actuar en consecuencia. Desde un punto de vista general, esto implica transformar datos visuales inicialmente no clasificados en representaciones procesadas, añadiendo información como pueden ser: etiquetas, localizaciones, contornos, relaciones espaciales o estimaciones de estado de los objetos o contextos iniciales planteados antes de la obtención de las imágenes. Este tipo de información permite su uso en múltiples ámbitos, entre los que destacan la robótica, el análisis de vídeo, la inspección industrial o los sistemas de asistencia a la conducción. Para llevar a cabo estos procesos se parte siempre de datos visuales, es decir, de una imagen digital[18]. Esta puede entenderse como una matriz de píxeles, donde cada elemento codifica información relacionada con la intensidad y el color.

Esta forma de representación introduce una serie de condicionantes técnicos que no pueden ignorarse. Algunos pueden ser la resolución de la imagen que limita el nivel de detalle que puede capturarse, mientras que el rango dinámico y la respuesta del sensor influyen directamente en la capacidad para registrar contrastes. Además, el ruido tanto de origen electrónico como derivado de condiciones de iluminación deficientes puede degradar de forma significativa la información obtenida por el sensor. En el caso del vídeo, a todo lo anterior se añade la dimensión temporal.

Factores como la frecuencia de muestreo (FPS), el desenfoque por movimiento o los procesos de compresión influyen en la coherencia de la información entre fotogramas consecutivos[19]. Por ello, cualquier sistema de Visión Artificial debe considerar no solo el modelo empleado, sino también la calidad y la naturaleza de los datos de entrada, ya que los errores sistemáticos presentes en la entrada tienden a propagarse a las etapas posteriores del proceso.

Por otra parte, la Visión Artificial ha experimentado una evolución, representada en la Tabla 2-2, que ha llevado esta tecnología desde enfoques tradicionales hacia técnicas basadas en *Deep Learning*. Los métodos clásicos se apoyaban en características diseñadas manualmente, como bordes, texturas o puntos de interés, combinadas con clasificadores relativamente simples. Esto implicaba una fuerte dependencia de la experiencia del diseñador y un esfuerzo considerable de reajuste ante cambios en el escenario.

Por el contrario, los enfoques modernos basados en redes neuronales aprenden representaciones jerárquicas directamente a partir de los datos, lo que ha permitido aumentar la resistencia frente a variaciones y mejorar el rendimiento en tareas complejas [20]. No obstante, esta mejora introduce nuevas exigencias, como la necesidad de grandes volúmenes de datos, un mayor coste computacional y una sensibilidad elevada cuando las condiciones reales difieren de las observadas durante el entrenamiento.

Aspecto	Métodos clásicos	Deep Learning (Modernos)
Extracción de características	Manual (bordes, texturas, SIFT)	Automática desde datos
Dependencia del diseñador	Alta (experiencia requerida)	Baja (aprende solo)
Robustez a variaciones	Baja (requiere reajustes)	Alta (jerarquías aprendidas)
Datos necesarios	Pocos	Grandes volúmenes
Coste computacional	Bajo	Alto (entrenamiento/inferencia)
Sensibilidad a condiciones reales	Moderada	Alta si difiere del entrenamiento
Rendimiento en tareas complejas	Limitado	Superior

Tabla 2-2 Tabla comparativa métodos clásicos frente a modernos[21]

El funcionamiento de un sistema de detección de objetos mediante Visión Artificial suele describirse mediante una serie de pasos normalmente estandarizados, que pueden representarse a través de un proceso compuesto por las siguientes fases como se muestra en la Figura 2-4.

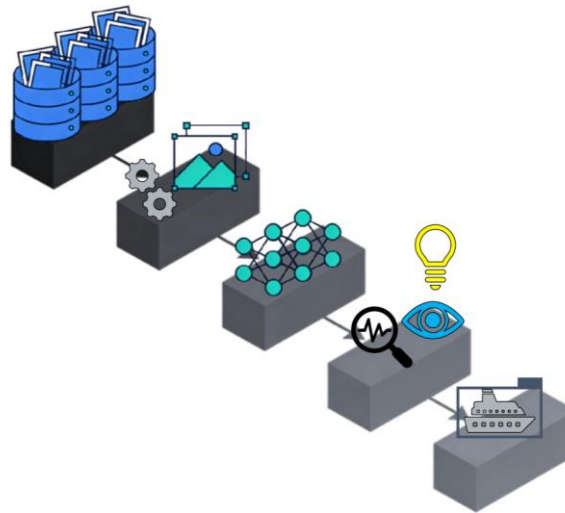


Figura 2-4 Funcionamiento de sistema de detección de objetos (Elaboración propia).

Las tareas que aborda la Visión Artificial pueden clasificarse en función del objetivo requerido [22]:

- En la función de clasificación, el objetivo consiste en asignar una o varias etiquetas generales a una imagen, respondiendo a preguntas preestablecidas, sin especificar la localización exacta de los elementos de la imagen.
- En localización y detección, además de identificar y clasificar, se incorpora información espacial, de tal modo que se busca localizar objetos individuales mediante áreas, normalmente empleando cajas delimitadoras asociadas a una clase y a un porcentaje de confianza.
- La segmentación incrementa aún más el nivel de detalle, al asignar una etiqueta a cada píxel de la imagen. En segmentación semántica, todos los píxeles pertenecientes a una misma clase se agrupan, mientras que en segmentación por instancias se diferencian objetos individuales incluso cuando pertenecen a la misma categoría.
- En escenarios dinámicos, el *tracking* añade coherencia temporal, permitiendo asociar un mismo elemento a lo largo de una secuencia de imágenes y estimar su trayectoria o estado.

La selección de la tarea adecuada depende directamente del objetivo del sistema: una inspección superficial puede resolverse mediante clasificación, mientras que tareas que implican interacción con el entorno, medición o navegación suelen requerir detección o segmentación [23].

En este contexto, los datos adquieren un papel fundamental. La calidad de un *dataset* no depende únicamente de su tamaño, sino también de la diversidad de escenarios, la representatividad, la coherencia de sus clasificaciones y el equilibrio entre clases, factores que condicionan directamente la capacidad de generalización del modelo. Es habitual encontrar problemas de sesgo o errores de clasificación que provocan que modelos con buen rendimiento aparente durante la etapa de validación resulten poco fiables en entornos reales.

En aplicaciones reales, la idoneidad de una solución no depende exclusivamente de su precisión. En sistemas que operan en tiempo real, aspectos como la latencia, el número de *frames* procesados por segundo, el consumo de memoria y la compatibilidad con el hardware disponible se convierten en requisitos esenciales. En consecuencia, es necesario encontrar un equilibrio entre exactitud y eficiencia, así como implementar mecanismos que garanticen robustez frente a condiciones no controladas, como cambios de iluminación, oclusiones parciales, variaciones de escala, fondos complejos o *drift*.

2.4 Detección de objetos con redes neuronales profundas

En la visión artificial, la detección de objetos se posiciona como una de sus aplicaciones principales y más impactantes. Esta técnica permite identificar, localizar y clasificar elementos específicos dentro de imágenes o vídeos en tiempo real, cuya función en la actualidad se utiliza en múltiples ámbitos tecnológicos como puede ser la asistencia a la conducción, el control de tráfico o la detección de errores y control de calidad en la industria. A continuación, se expondrán los diferentes modelos para la detección y clasificación de objetos que existen, cada uno con sus características y usos principales.

2.4.1 *Faster R-CNN*

Se caracteriza por ser un modelo de dos etapas centrado en crear regiones y luego clasificar y pulir los objetos en cajas no obstante necesitaba una menor cantidad de *FPS* para tener esa calidad por lo que su uso era más bien en escenas estáticas. Se trata de un modelo bastante usado en investigación y en *frameworks* como puede ser Detectron2 usado en Meta que, además también incluía versiones de RetinaNet [24].

2.4.2 *RetinaNet*

Se trata un modelo de detección de objetos de una sola etapa que revolucionó el campo al combinar alta precisión con simplicidad. Lanzado en 2017, se ideó para superar las limitaciones de detectores previos como YOLO o SSD, ofreciendo resultados competitivos frente a arquitecturas de dos etapas sin sacrificar velocidad. Su popularidad viene de la Focal Loss, que sirve para tareas como vigilancia marítima o facial. La arquitectura de RetinaNet parte de un *backbone* que procesa la imagen de entrada para extraer características jerárquicas. Sobre este se superpone una *Feature Pyramid Network* (FPN), que fusiona características de baja y alta resolución en una pirámide multiescala. Esto permite detectar objetos desde los más pequeños (en mapas de alta resolución) hasta los más grandes (en mapas de baja resolución), todo en una única pasada optimizando tanto precisión como eficiencia [25].

2.4.3 *SSD*

SSD (*Single Shot MultiBox Detector*) es un detector de objetos de una sola etapa propuesto en 2016 por Google como alternativa rápida a YOLO. Predice clases y cajas delimitadoras directamente en una única pasada sobre la imagen, priorizando velocidad para aplicaciones en tiempo real.

Su arquitectura emplea un *backbone* convolucional que genera mapas de características a múltiples escalas. Sobre estos se colocan default boxes (análogos a *anchors*) con distintas proporciones, prediciendo simultáneamente clases y ajustes de caja, seguidos de NMS para filtrar detecciones. MobileNet-SSD optimiza SSD para dispositivos de recursos limitados, reemplazando VGG por MobileNet (convoluciones separables por profundidad). Reduce drásticamente tamaño 4 a 20 MB y consumo computacional, manteniendo velocidad de 20-30 FPS en dispositivos móviles [26].

2.4.4 *YOLO*

YOLO, algoritmo líder en detección de objetos y segmentación de imágenes, surgió en 2015 de la mano de Joseph Redmon y Ali Farhadi en la Universidad de Washington. Su rápido ascenso se debe a la notable combinación de velocidad de procesamiento y exactitud en los resultados.

Fue concebido originalmente como un detector unificado que predice cajas y probabilidades de clase directamente a partir de la imagen completa en una sola evaluación de la imagen, lo que explica su orientación a aplicaciones en tiempo real. Yolo ha sido la referencia en detección de objetos en tiempo real durante la última década [27].

La selección de la arquitectura YOLO para el desarrollo de este proyecto se fundamenta en su capacidad para ofrecer un equilibrio óptimo entre precisión y velocidad de procesamiento en entornos dinámicos como son los escenarios marítimos. A diferencia de los modelos de dos etapas como Faster R-CNN, que priorizan la precisión debido a una menor tasa de *FPS*, YOLO emplea un proceso de una sola etapa que evalúa la imagen en una única pasada. Si bien alternativas como SSD resultan eficientes para dispositivos móviles y RetinaNet destaca en escenas densas mediante la *Focal Loss*, YOLO ha sido elegida la solución idónea para este tipo de ámbito además de estar ampliamente referenciada y poseer una gran cantidad de recursos y documentación al respecto.

2.5 Detección de objetos con Yolo

El sistema YOLO se ha convertido en uno de los modelos más influyentes en el ámbito de la detección de objetos. Desde su aparición, ha evolucionado a través de distintas versiones que han mejorado su precisión, velocidad y capacidad de detección. En este apartado se presentará su origen, las principales versiones y características.

2.5.1 Funcionamiento de YOLO

YOLO es uno de los enfoques más representativos en detección de objetos, ya que replantea el problema como una única tarea de regresión. Para comprender su eficiencia en el procesamiento de imágenes y generación de predicciones en aplicaciones en tiempo real, es fundamental analizar su funcionamiento.

El *workflow* de YOLO [28] en detección de objetos busca un resultado eficiente véase Figura 2-5, procesando los datos de entrada en este caso “la imagen” en una sola vez a través de una red neuronal convolucional. En primera instancia, la imagen de entrada se subdivide en una rejilla de tamaño estandarizado $P \times P$, donde cada subárea de la imagen asume la tarea de detectar los posibles objetos cuyo centroide se encuentra dentro de ella. Para cada celda, YOLO predice un conjunto de cajas delimitadoras, junto con un porcentaje de confianza el cual muestra la probabilidad de que sea un objeto en específico y qué tan preciso es la localización que da el modelo. Por otra parte, cada celda tiene almacenada las probabilidades asociadas a los distintos tipos de clasificación, lo que permite identificar de forma más exacta el tipo de objeto presente en cada área o en la imagen global pudiendo ser, por ejemplo: personas, vehículos u otros elementos de interés.

Cada caja delimitadora se grafica mediante un conjunto de parámetros en los que se incluyen las coordenadas del centro (x,y) con sus dimensiones de ancho (w) y alto (h), todos ellos estandarizados respecto al tamaño de la imagen. Las suposiciones se obtienen directamente como salida de la red neuronal convolucional, la cual aprende a trazar un mapa con los píxeles de la imagen y transfórmalos a coordenadas espaciales y categorizar los objetos, transformando la detección en un problema de regresión. Cuando ya están generadas todas las predicciones posibles, YOLO crea múltiples cajas que pueden solaparse o no entre sí para un mismo objeto. Para solventar esta cuestión, se aplica el algoritmo de *Non-Maximum Suppression* (NMS), cuyo objetivo es eliminar las cajas redundantes y conservar únicamente aquellas con mayor nivel de confianza [29].

Finalmente, tras aplicar este filtrado, YOLO devuelve como resultado un conjunto reducido de cajas delimitadoras finales, cada una acompañada de su etiqueta de clase correspondiente y su valor de confianza. Gracias a esta arquitectura unificada y al hecho de que toda la imagen se analiza simultáneamente, YOLO logra una detección de objetos rápida y eficaz, lo que lo convierte en una solución especialmente adecuada para aplicaciones en tiempo real como el análisis de vídeo, la vigilancia o los sistemas de conducción autónoma [30].

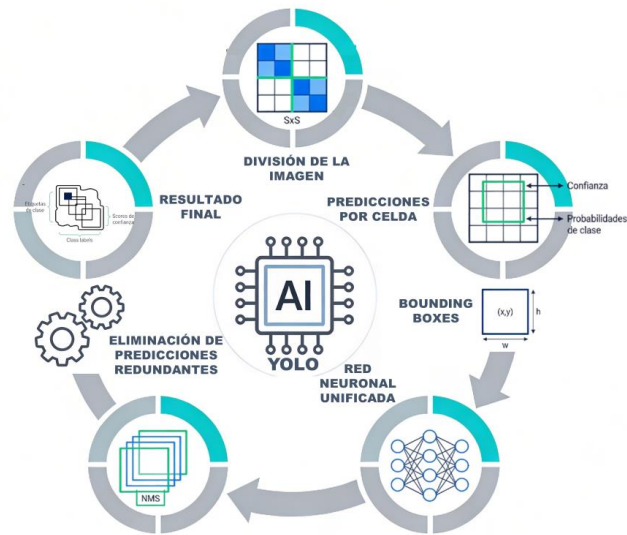


Figura 2-5 Esquema funcionamiento de YOLO (Elaboración propia)

2.5.2 Versiones y evolución histórica

Las sucesivas versiones y sus características se encuentran expuestas en la Tabla 2-3. A continuación, se describirán en detalle sus puntos fuertes y limitaciones; además se proporcionará una justificación a la selección de la versión de Yolo.

Versión	Año	Innovaciones Principales	Aplicaciones Destacadas
YOLOv1	2015	Regresión única, one-stage	Detección básica tiempo real
YOLOv2	2016	Normalización por lotes, cajas de anclaje, clústeres	Mejora precisión inicial
YOLOv3	2018	Backbone eficiente (Darknet-53), FPN	Objetos multi-escala
YOLOv4	2020	Aumentación Mosaic, sin anclaje, nueva pérdida	Aumentación de datos avanzada
YOLOv5	2020	Hiperparámetros automáticos, export multi-formato	Prototipos rápidos
YOLOv6	2022	Optimizado para edge	Robots autónomos
YOLOv7	2022	Estimación de pose (puntos clave COCO)	Detección + estimación de pose
YOLOv8	2023	Multi-tarea (detección/segmentación/clasificación)	Visión general
YOLOv9	2024	PGI, arquitectura GELAN	Precisión en datasets complejos
YOLOv10	2024	Extremo a extremo sin NMS	Tiempo real sin post-proceso
YOLOv11	2024	Detección/segmentación/pose/seg tracking/clasificación	Tareas versátiles múltiples
YOLOv26	2026	Despliegue edge sin NMS	Aplicaciones IoT

Tabla 2-3 Comparativa versiones de YOLO (Extraído de Ultralytics)[31]

YOLO propuso desde su primera versión tratar la detección como un único problema de regresión: analiza la imagen una sola vez y predecir coordenadas y clases. Era rápido, pero tenía problemas serios detectando objetos pequeños o agrupados.

Las versiones siguientes, YOLOv2 y YOLOv3, solucionaron gran parte de estos fallos introduciendo conceptos como *batch normalization*, resoluciones dinámicas y, en específico, el uso de redes backbone más profundas (Darknet-19 y Darknet-53) [32]. Sin embargo, a medida que ganaban precisión, también aumentaban su carga computacional, alejándose un poco de la posibilidad de correr en *hardware* más limitado. El verdadero cambio de dogma hacia la eficiencia llegó con YOLOv4 y YOLOv5. Estas versiones se centraron no solo en las redes neuronales, sino en las técnicas de (*bag of freebies*) que consiste en una serie de modificaciones que da lugar a un aumento de datos y entrenamiento que mejoran la precisión sin coste en inferencia [33], [34]. A partir de esta versión se empieza a seguir una estandarización clara de variantes por tamaño. Poco después, aparecieron modelos experimentales como YOLOv6 y v7, cada uno aportando mejoras poco perceptivas en la estructura de convolución [35], [36].

Sin embargo, la versión más característica y popular es YOLOv8. Esta versión representa un salto cualitativo importante por ser *anchor-free*. Al eliminar la dependencia de las "cajas de anclaje" predefinidas, el modelo reduce drásticamente la cantidad de cálculos necesarios en la etapa final de predicción (el cabezal o *head*), lo que simplifica el proceso de *Non-Maximum Suppression* (NMS). Además, su arquitectura desacoplada permite aprender por separado la clasificación del objeto y su localización, mejorando la convergencia. Las versiones más recientes de la serie YOLO continúan perfeccionando los avances introducidos en YOLOv8, moviéndose hacia una mayor estabilidad, eficiencia y precisión [37]. En este contexto, YOLOv9 aborda uno de los principales retos de las redes neuronales convolucionales: el cuello de botella de información y la degradación del gradiente. Para ello incorpora dos elementos fundamentales: el *Programmable Gradient Information* (PGI), que preserva la información relevante a lo largo de las capas y mejora la estabilidad del entrenamiento; y la *Generalized Efficient Layer Aggregation Network* (GELAN). El resultado es un modelo más preciso y eficiente que YOLOv8, con mejor aprovechamiento de los parámetros y una alta adaptabilidad tanto en servidores como en dispositivos de bajos recursos [38].

Poco tiempo después con YOLOv10 se avanza hacia un tipo de detección denominada *end-to-end* al suprimir la obligación de pasar por NMS, que es normalmente una de las etapas más demandantes del flujo de inferencia. Valiéndose de un esquema de entrenamiento cuya función principal son los *dual assignments*, el modelo obtiene la habilidad de generar predicciones precisas, logrando un sistema libre de NMS que minimiza el retardo y optimiza el proceso de inferencia. Además, de incorporar una cabecera de clasificación más ligera, lo que minimiza el número de parámetros y operaciones sin comprometer la precisión. Estas mejoras se traducen en incrementos de precisión y reducciones significativas en parámetros y tiempo de inferencia en todos los tamaños de modelo [39].

La versión YOLOv11 refuerza la arquitectura del *backbone*. Su diseño amplía las capacidades de la red hacia múltiples tareas, como detección, segmentación, clasificación, estimación de pose y *Oriented Bounding Boxes* (OBB). Además, mantiene una estructura *anchor-free* que mantiene un balance entre eficacia, velocidad y precisión, garantizando un rendimiento óptimo [40].

Finalmente, YOLO26 constituye un modelo esencial para lograr la simplificación general. En esta última versión se elimina en su totalidad la *Distribution Focal Loss* y NMS, impulsando un modelo de flujo *end-to-end* más ágil y eficaz consolidándose como una de las versiones más eficientes para producción en tiempo real [41].

En todas sus versiones, YOLO se distribuye en una escala de cinco tamaños según la cantidad de parámetros (capas y neuronas) que componen la red [42]:

- Nano (n): La versión minimalista, optimizada al extremo para dispositivos de borde (*edge*).
- Small (s): Un paso intermedio ligero.

- Medium (m), Large (l) y Extra Large (x): Modelos masivos diseñados para estaciones de trabajo con GPUs dedicadas de alto rendimiento, donde el consumo energético no es una restricción.

El entorno móvil presenta desafíos de ingeniería que no existen en un PC de escritorio:

- Potencia de cómputo limitada: Los procesadores móviles y sus gráficas integradas no pueden mover millones de parámetros a 30 o 60 FPS sin saturarse. Intentar correr un modelo "Medium" resultaría en una tasa de refresco inaceptable provocando lag y arruinando la experiencia de uso en vivo.
- Eficiencia térmica y energética: Un modelo pesado obliga a la CPU/GPU a trabajar al 100% constantemente. Esto provoca dos problemas críticos en un dispositivo móvil: el *thermal throttling* provocando que el móvil se recalienta y baja su velocidad para protegerse y el drenaje acelerado de la batería.
- Tamaño: La variante Nano pesa normalmente apenas unos 6 MB, frente a los cientos de MB de las versiones grandes. Esto es crucial para que las aplicaciones sean ligeras y el modelo pueda cargarse rápidamente en la memoria RAM de un teléfono.

Por lo tanto, la opción seleccionada para este proyecto es YOLOv11 en su variante Nano, debido a su versatilidad, reducido peso computacional y adecuación para su ejecución en dispositivos Android. Asimismo, esta elección permite minimizar las limitaciones y problemas asociados al uso de versiones de mayor tamaño, las cuales podrían comprometer el rendimiento del sistema en entornos de baja potencia, un aspecto especialmente relevante en aplicaciones que requieren rapidez y eficiencia. Respecto a la versión 26, lanzada muy recientemente, YOLOv11 cuenta con un ecosistema más maduro y una fiabilidad más contrastada.

2.6 Conjunto de datos

El desarrollo de este proyecto de visión artificial basados en Redes Neuronales Convolucionales usando YOLO no depende únicamente de la topología de la red, sino de la representatividad y calidad de los datos suministrados. La selección del conjunto de datos de entrenamiento define los límites operativos del sistema.

En la visión artificial, los *datasets* no son simples bancos de imágenes, sino la materia prima que permite al modelo construir su filtro de características. Para que un algoritmo como YOLO minimice su función de pérdida, requiere ejemplos que cubran la varianza intra-clase que se define como diferentes tipos de barcos y la varianza inter-clase por ejemplo barcos vs boyas o islas [43].

Se diferencian dos categorías de conjuntos de datos necesarios para entrenar una IA:

- Datasets de "Pre-entrenamiento" (*Benchmark*): Conjuntos masivos diseñados para que la red aprenda características visuales de bajo y medio nivel (bordes, texturas, formas geométricas básicas).
- Datasets de "Ajuste Fino" (*Downstream Task*): Conjuntos curados que adaptan el conocimiento previo al dominio específico de aplicación, en este caso, el entorno marítimo.

Prosiguiendo con la anterior clasificación se procederá a explicar en los siguientes apartados los principales conjuntos de datos que existen en el ámbito de la detección de objetos dentro del contexto marítimo.

2.6.1 COCO

El dataset COCO (*Common Objects in Context*) representa el estándar para la evaluación de modelos de detección de objetos. A diferencia de sus predecesores como PASCAL VOC o ImageNet. COCO contiene más de 330.000 imágenes, de las cuales más de 200.000 están etiquetadas, sumando 1.5 millones de instancias de objetos repartidas en 80 categorías semánticas. Fue diseñado para abordar el

problema del reconocimiento no icónico, es decir, objetos que no ocupan el centro de la imagen ni se presentan en condiciones de estudio, sino inmersos en escenas complejas y desordenadas [44].

Sin embargo, tiene sus limitaciones ya que, a pesar de su robustez, COCO presenta un "sesgo de dominio" hacia entornos terrestres y urbanos como se aprecia en la Figura 2-6 [45]. Aunque incluye la clase "boat" las instancias suelen ser embarcaciones recreativas en contextos de playa o puerto deportivo, con fondos de alta frecuencia visual (edificios, árboles). Un modelo entrenado exclusivamente en COCO suele fallar al desplegarse en mar abierto debido a la homogeneidad de la textura del agua, los reflejos especulares y la falta de puntos de referencia visuales, lo que genera falsos positivos en las olas o nubes

Sin embargo, su valor no reside en la cantidad bruta, sino en la complejidad de sus anotaciones. COCO introduce métricas de evaluación estrictas. Esta métrica penaliza severamente a los detectores que localizan el objeto, pero no ajustan la caja delimitadora con precisión milimétrica. Lo que deriva en que entrenar o validar inicialmente en COCO garantiza que la arquitectura YOLO seleccionada es capaz de gestionar [46]:

- Oclusiones Parciales: Objetos tapados parcialmente por otros.
- Variación de Escala: COCO divide explícitamente los objetos en *small*, *medium* y *large*. Esta capacidad es transferible directamente a la detección de barcos, que a menudo aparecen como pequeños puntos en el horizonte. También cabe destacar que existen modelos de YOLO ya preentrenados con este conjunto de datos directamente ofrecidos por YOLO.

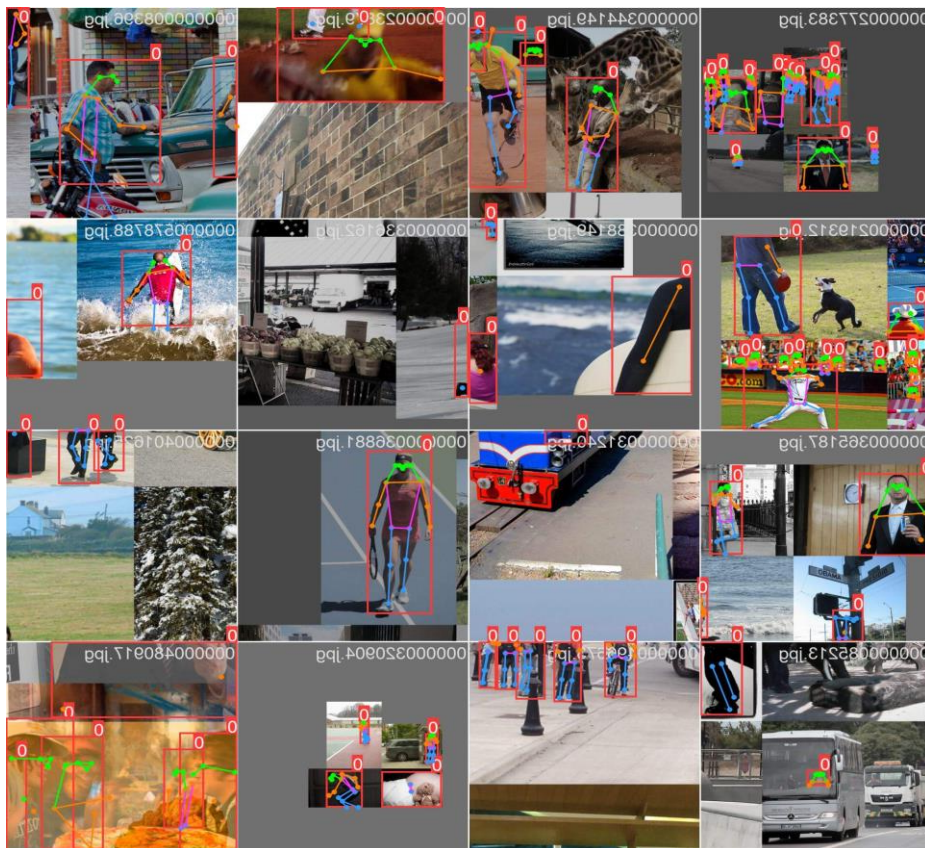


Figura 2-6 Dataset COCO [45]

2.6.2 Marvel

MARVEL está posicionado como un *dataset* de gran escala orientado al reconocimiento visual de grano fino que consta de más de 2 millones de imágenes que cubren 197.430 buques únicos. A diferencia de *datasets* contruidos en laboratorio, MARVEL se nutre de imágenes recolectadas por comunidades de *shipspotting*, lo que asegura que las imágenes capturan las condiciones operativas reales de los buques. La característica más crítica de MARVEL es su validación cruzada mediante AIS (*Automatic Identification System*). Cada imagen está asociada a metadatos verificados que incluyen el número IMO (*International Maritime Organization*) y el MMSI, lo que proporciona un proceso de alta fiabilidad que elimina el ruido de etiquetado común en otros *datasets* donde voluntarios no expertos confunden, por ejemplo, un buque de carga general con un granelero. Otra diferencia entre estos *datasets* es que mientras que COCO agrupa todo bajo una única etiqueta como puede ser “barco” MARVEL [47] implementa una jerarquía de clases basada en la funcionalidad del buque a clasificar. El *dataset* original define 109 clases base que se pueden agrupar en subcategorías como puede ser *Tanker* o *Carrier* como se observa en la Figura 2-7.



Figura 2-7 Dataset Marvel [47]

2.6.3 SeaShips

SeaShips es un amplio conjunto de datos de imágenes diseñado específicamente para el entrenamiento y la evaluación modelos de detección de barcos mediante inteligencia artificial. Creado a partir de aproximadamente 10.080 *frames* de video del mundo real capturados por cámaras de vigilancia costera, el conjunto de datos actual incluye 31.455 imágenes de alta resolución. Todas estas imágenes están anotadas con gran precisión mediante etiquetas y cajas delimitadoras (*bounding boxes*), cubriendo diferentes variaciones de escala, iluminación, punto de vista u oclusiones. A diferencia de otros *datasets* más generalistas, estas variaciones en sus parámetros y condiciones diferentes a las ideales sirven para entrenar a los modelos de visión artificial de una forma más compleja y realista. Los modelos entrenados con *datasets* genéricos suelen fallar en escenarios reales de mar abierto. Sin embargo, los algoritmos entrenados con SeaShips logran mantener una alta precisión incluso cuando se evalúan con imágenes de otras bases de datos o en entornos portuarios reales con fondos complejos [48].

El dataset clasifica los barcos en seis categorías principales y comunes como se aprecia en la Figura 2-8: cargueros de mineral (*Ore Carrier*), graneleros (*Bulk Carrier*), buques de carga general (*General*

Cargo Ship), portacontenedores (*Container Ship*), barcos de pesca (*Fishing Boat*) y barcos de pasajeros (*Passenger Ship*).

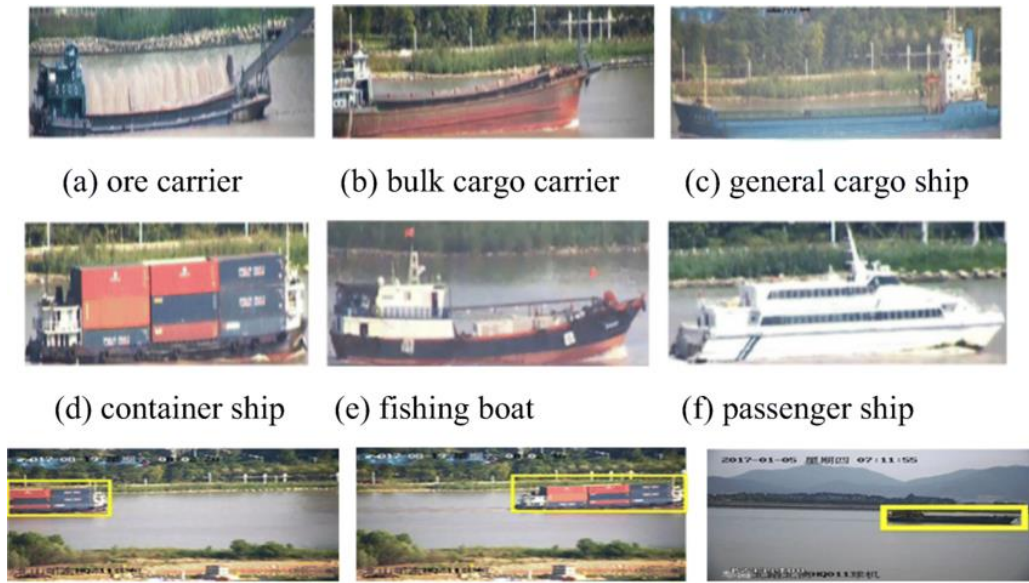


Figura 2-8 Dataset SeaShips [48]

2.6.4 Datasets de imágenes aéreas

ShipRSImageNet destaca por ser el *dataset* más grande con clasificación específica de tipos de buque, combinando imágenes de 6 satélites chinos de alta resolución junto con Google Earth. Contiene 3.435 imágenes de alta resolución con 17.573 buques anotados mediante *bounding boxes* orientados, distribuidos en 50 clases en los que se sabe que incluyen portaaviones, destructores, fragatas, submarinos, cargueros, petroleros, contenedores y pesqueros. Esto produce una capacidad de identificar específicamente el tipo de embarcación lo que facilita el análisis y clasificación de los buques en los lugares de alta densidad de tráfico marítimo [49].

MASATI Dataset se enfoca en escenarios marítimos dinámicos reales capturados desde plataformas aéreas como son los drones y helicópteros sobre el mar Mediterráneo, con aproximadamente 8.000 imágenes. Clasifica los buques en 7 clases: barcos simples, multi-barcos, barcos costa, barcos por tamaño (pequeño, mediano, grande) y combinaciones costa-barco, además incluye oleaje real, espuma, reflejos solares y barcos en movimiento como se puede observar en la Figura 2-9 [50].

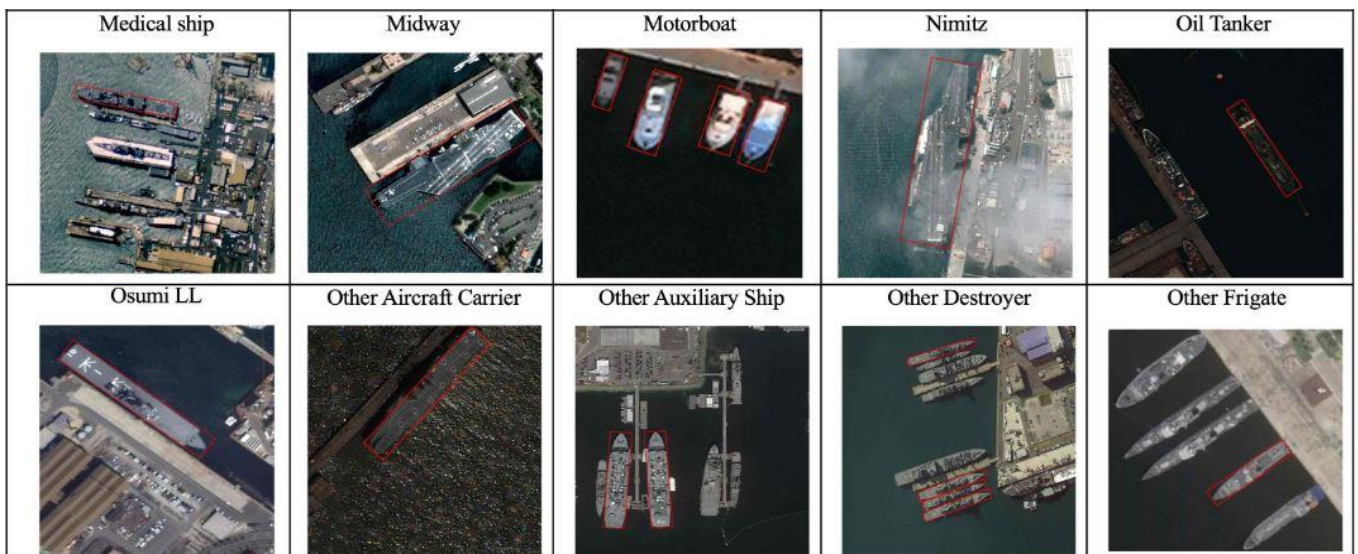


Figura 2-9 Datasets aéreos [49]

2.7 Entorno de trabajo

Para llevar a cabo un proyecto de estas características, resulta indispensable contar con un entorno de desarrollo apropiado. En este apartado se exponen diversas alternativas que cumplen con los requisitos exigidos, tales como disponer de un conjunto de herramientas flexibles, soporte continuado y tecnologías ampliamente probadas, elementos fundamentales para alcanzar los objetivos previstos en tiempo y forma. Actualmente, existen múltiples opciones para el desarrollo de aplicaciones gráficas interactivas y videojuegos, entre las que destacan Unreal Engine, Android Studio y Unity véase Figura 2-10.

2.7.1 Unreal Engine

Unreal Engine es un motor gráfico focalizado en videojuegos desarrollado por Epic Games, es reconocido ampliamente por poder generar gráficos hiperrealistas y físicas complejas con un alto nivel de detalle. Este motor funciona con C++, es la herramienta normalmente usada para desarrollar juegos AAA, simulaciones complejas y producciones cinematográficas. Además, integra tecnologías novedosas como son la iluminación avanzada y el raytracing para ofrecer experiencias visuales atractivas [51].

2.7.2 Android Studio

Android Studio es el entorno de desarrollo oficial desarrollado por Google para crear aplicaciones orientadas a Android. Este programa integra un editor de código flexible, además de un emulador de dispositivos de gran tamaño y herramientas de depuración que clarifican, ayudan y además resuelven errores en el proceso de creación de código con lenguajes como Java y Kotlin. Cabe destacar que posee una amplia lista de bibliotecas, plantillas, herramientas para el posterior ensayo y puesta a punto de los proyectos, los cuales ayudan a los desarrolladores a acelerar su flujo de trabajo y a crear aplicaciones funcionales y optimizadas desde el inicio [52].

2.7.3 Unity

Unity es un motor de desarrollo multiplataforma ampliamente utilizado que reúne en un solo entorno todas las herramientas necesarias para crear videojuegos y experiencias interactivas en 2D y 3D. Se caracteriza por su gran flexibilidad y facilidad de uso, lo que lo ha convertido en una opción muy popular dentro de la industria, especialmente en el desarrollo de juegos móviles y proyectos independientes. A través de su editor visual intuitivo y el uso de programación mediante scripts, los desarrolladores pueden añadir sistemas de físicas, animaciones o inteligencia artificial, y además exportar sus proyectos a una gran variedad de dispositivos y plataformas [53].

En este sentido, cabe destacar que el entorno de desarrollo seleccionado para este proyecto es Unity. Esta decisión se fundamenta en su portabilidad y en su continuidad con la línea de investigación de la que deriva este proyecto, lo cual facilita su integración futura. Además, se trata de una herramienta ampliamente consolidada en el sector, que ofrece un soporte para la integración de modelos de inteligencia artificial y sistemas de detección de objetos.



Figura 2-10 Logos de Unity, Unreal Engine y Android Studio [51],[52],[53]

3 DESARROLLO DEL TFG

En el presente capítulo, después de haber realizado una contextualización de la visión artificial y los modelos de detección de objetos. A continuación, se describirá el entorno de desarrollo de la aplicación. También se explicará el proceso de creación, la implementación de mejoras y su funcionamiento general.

3.1 Dispositivos empleados

Para la realización de este trabajo ha sido necesario el uso de diversos dispositivos: el dispositivo de edición /desarrollo del proyecto y un dispositivo móvil, el cual sirva de herramienta de despliegue para su implementación portátil. Por otra parte, cabe resaltar el uso de un servidor para el entrenamiento de un modelo propio.

3.1.1 Ordenador

El ordenador proporcionado por el Centro Universitario de la Defensa, modelo Asus GL502VS (Figura 3-1) con las características expuestas en la Tabla 3-1.



Figura 3-1 Asus GL502VS (Elaboración propia)

Componentes	Características
Procesador	Intel Core i7-7700HQ hasta 2,8 GHz
RAM	32GB DDR4
Gráfica	NVIDIA GeForce GTX 1070 con 8 GB GDDR5

Tabla 3-1 Características Asus GL502VS

3.1.2 Dispositivo móvil

El teléfono inteligente proporcionado por el Centro Universitario de la Defensa, modelo Xiaomi Redmi Note 11 (Figura 3-2) con las características expuestas en la Tabla 3-2.



Figura 3-2 Xiaomi Redmi Note 11 (Elaboración propia)

Componentes	Características
Procesador	Qualcomm Snapdragon 680 (8 núcleos, hasta 2,4 GHz, 6 nm)
RAM	6 GB LPDDR4X
Gráfica	Qualcomm Adreno 610

Tabla 3-2 Características Xiaomi Redmi Note 11

3.2 Configuración general de Unity

El motor gráfico seleccionado para la realización de este proyecto es Unity, el cual utiliza el lenguaje de programación en C#. A continuación, se procederá a explicar el entorno del proyecto y la configuración de Unity adecuada para la creación y posterior mejora de la aplicación en dispositivos móviles. Una vez definido Unity como el entorno seleccionado para la aplicación, cabe precisar que se ha utilizado la versión 6000.3.9f1, por ser la más reciente disponible durante el desarrollo de este proyecto.

El proceso de configuración del entorno de trabajo comienza con la descarga e instalación de Unity Hub, el programa oficial para la gestión de proyectos y versiones del motor gráfico. Una vez instalado, se procede a seleccionar y descargar la versión de Unity requerida. Durante este mismo paso, resulta indispensable instalar los módulos adicionales necesarios para la futura exportación de la aplicación a sistemas operativos móviles, como Android o iOS (véase Figura 3-3 Entorno Unity Hub Figura 3-3). De manera simultánea, se incluye la instalación del entorno de desarrollo Microsoft Visual Studio

Community 2022. Este software proporciona las herramientas necesarias para redactar y editar los *scripts* del proyecto, destacando por su capacidad de sincronización directa con Unity, lo que permite compilar y aplicar los cambios en el código de forma automática e inmediata cada vez que se guarda el archivo.

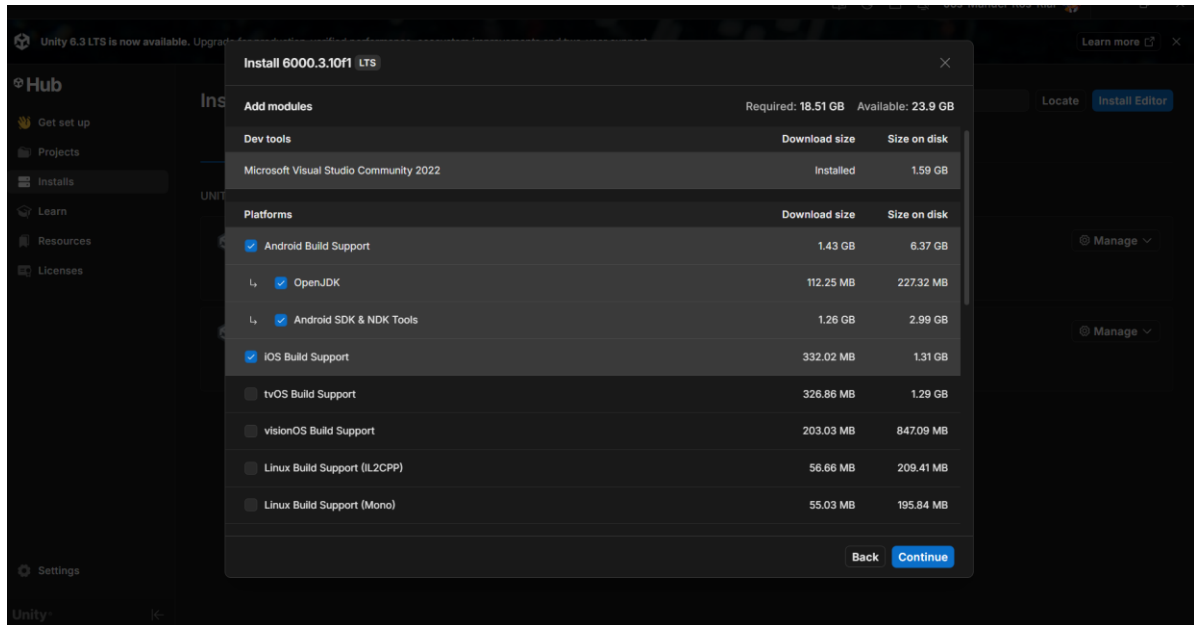


Figura 3-3 Entorno Unity Hub (Elaboración propia)

Al iniciar por primera vez el proyecto, Unity crea una serie de objetos predeterminados como son *Main camera* y *Directional light*, que son innecesarias para este trabajo y se deben eliminar antes de comenzar a trabajar en el entorno de desarrollo. Para mantener unas buenas prácticas y que el entorno se encuentre lo más limpio y ordenado posible, además se procederá a cambiar la distribución de la interfaz al modo 2 by 3. En la interfaz principal se distinguen los paneles de Unity: la vista de Escena, el Simulador (o vista de Juego), el Inspector y la Jerarquía. Asimismo, en la ventana de Proyecto se ha estructurado el contenido en dos directorios principales: una carpeta de recursos generales y otra denominada Escenas. Dentro de la carpeta de recursos se subdividirá en 4 clases: códigos, imágenes, modelos onnx y música véase (Figura 3-4).

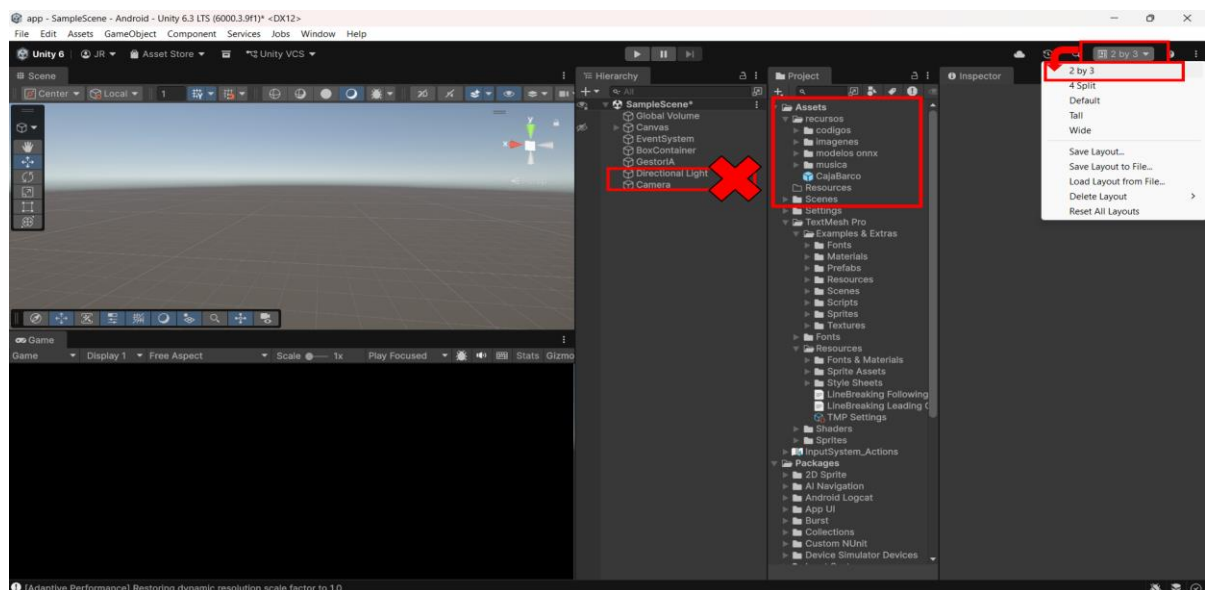


Figura 3-4 Escena general de Unity (Elaboración propia)

A continuación, descargaremos e instalaremos un paquete necesario para el manejo de inteligencia artificial en Unity, denominado Sentis. Para ello iremos a la ventana de Windows y en el desplegable bajaremos hasta *Package Management* y en el desplegable secundario otra vez en *Package Management*. Abierto el gestor, clicamos en el desplegable que se expone en la Figura 3-5 y seleccionamos la opción (Install package by name) e introducimos (com.unity.ai.inference).

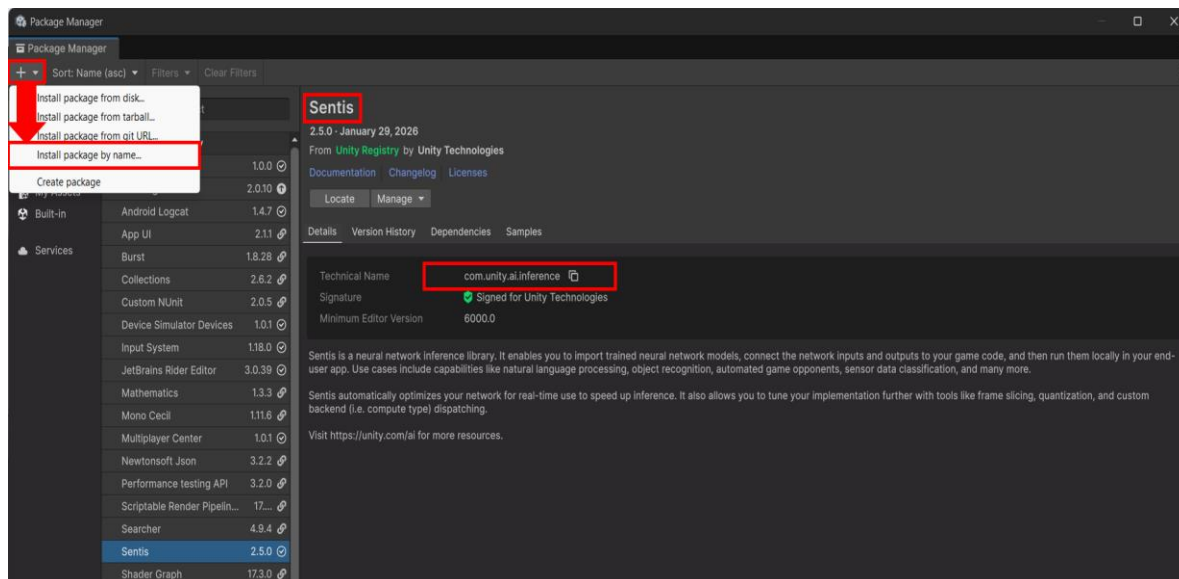


Figura 3-5 Menú Package Management (Elaboración propia)

3.3 Funcionalidades de la aplicación

Introducidas las funcionalidades o requisitos de la aplicación, en este apartado se describe en detalle su implementación empleando el entorno de desarrollo configurado previamente.

Durante el desarrollo de esta aplicación, el sistema ha sido estructurado mediante un diseño modular con el fin de resolver los problemas técnicos inherentes al procesamiento de visión artificial en dispositivos móviles. Dado que este tipo de plataformas presenta limitaciones de hardware, ha sido necesario implementar soluciones específicas de optimización. En el presente apartado se detallan las funcionalidades principales del sistema, tanto aquellas destinadas a mitigar dichas restricciones técnicas como las orientadas a perfeccionar la experiencia final del usuario.

Por lo tanto, estas funcionalidades se clasificarán en cinco secciones:

1. Integración del Sistema: Unity, Sentis y YOLO

El núcleo de la aplicación radica en la integración entre el entorno de renderizado gráfico con Unity y el procesamiento de visión artificial mediante YOLO. Para lograr este enlace, se emplea el paquete nativo Unity Sentis actualmente denominado *Inference Engine*.

Preprocesamiento de datos: Las redes neuronales requieren que la información de entrada esté estructurada en matrices numéricas multidimensionales es decir tensores. La aplicación usa vídeo en tiempo real desde la cámara del dispositivo, aplica las transformaciones espaciales necesarias (recorte y escalado) y convierte la imagen resultante en un tensor de 640x640 píxeles.

Ejecución Asíncrona: Para evitar el bloqueo del hilo principal de la aplicación, la ejecución del modelo se realiza de forma asíncrona. El Tensor de entrada se envía a un proceso en segundo plano es decir un *worker*, lo que permite que el motor gráfico mantenga una tasa de refresco estable mientras el modelo completa la inferencia *IsReadbackRequestDone* y devuelve los resultados.

2. Modelo de VisiónArtificial: Arquitectura YOLO

La aplicación ejecuta la arquitectura de detección de objetos YOLO, seleccionada por su eficiencia y viabilidad para la detección en tiempo real sobre *hardware* móvil como se justificó previamente.

Especialización del *dataset*: A diferencia de los modelos genéricos entrenados con conjuntos de datos como COCO que se usó en su primera versión, este modelo ha sido entrenado a un ámbito marítimo con un subconjunto de SeaShips. Esta especialización permite al sistema clasificar con alta precisión seis categorías de buques: *Bulk Cargo Carrier*, *Container Ship*, *Fishing Boat*, *General Cargo Ship*, *Ore Carrier* y *Passenger Ship*.

Salida del modelo: Como resultado de la inferencia, la red neuronal no devuelve una clasificación única, sino un conjunto de predicciones. Este incluye múltiples coordenadas de cajas delimitadoras, un nivel de confianza y la clase asignada a cada detección.

3. Filtrado de Detecciones: Algoritmo NMS

A raíz de cómo funciona la arquitectura YOLO, es habitual la generación de predicciones redundantes, donde un mismo objeto es enmarcado por múltiples cajas delimitadoras superpuestas.

Para reducir este ruido, el sistema implementa el algoritmo de Supresión de No Máximos. Este método evalúa la métrica de Intersección sobre Unión (IoU) entre cajas adyacentes. Si el solapamiento supera un umbral configurable predefinido *iouThreshold*, el sistema deduce que ambas predicciones corresponden a la misma embarcación, procediendo a descartar la predicción con menor índice de confianza y conservando únicamente la estimación más precisa.

4. Estabilidad: Seguimiento (*Tracking*)

Los modelos de detección puros carecen de contexto temporal, evaluando cada fotograma de manera independiente. Esto genera inestabilidad visual o *flickering* en la interfaz, ya que en caso de que el modelo falle durante una fracción de segundo se perdería el seguimiento. Para garantizar una experiencia visual estable, se ha implementado un algoritmo de seguimiento (*tracking*) personalizado:

Identidad: El sistema asigna un identificador único (*TrackedObject*) a cada embarcación detectada, manteniendo un registro de los objetos presentes en la escena.

Asociación Euclidiana: En cada nuevo fotograma, se calcula la distancia espacial entre las posiciones guardadas en la detección anterior y las nuevas. Si un nuevo objeto se encuentra dentro de un margen de distancia (*maxTrackingDistance*), el sistema asume la continuidad del objeto en movimiento.

Tolerancia a fallos: Se ha introducido un margen de memoria temporal (*maxLostFrames*). Ante la pérdida momentánea de una detección que puede ser causada por reflejos u oclusiones, el sistema retiene el objeto en memoria durante un número determinado de fotogramas antes de descartarlo definitivamente.

5. Optimización de Rendimiento y Experiencia de Usuario (UX)

El software incorpora estrategias específicas de optimización para garantizar un rendimiento fluido y eficiente energéticamente en dispositivos Android:

Intervalo de detección: El procesamiento llevado por YOLO conlleva un gasto de recursos alto. Para minimizar la carga sobre el procesador, se ha introducido un intervalo de detección (*detectEveryNFrames*). Mediante esta función, el sistema se actualiza constantemente, pero el modelo solo evalúa una fracción de los fotogramas, logrando un balance óptimo entre rendimiento y precisión.

Zoom Digital por *software*: Se ha implementado una función de escalado digital. Alterando las coordenadas de la textura de entrada previa a su conversión en tensor, el modelo analiza un área ampliada del centro de la pantalla, incrementando la capacidad del sistema para detectar embarcaciones ubicadas a mayores distancias.

3.4 Desarrollo de la aplicación

En Unity, una escena funciona como un escenario o nivel individual donde se organizan e interactúan todos los elementos visuales, objetos y scripts que conforman una parte específica del proyecto. Bajo este concepto, el desarrollo de esta aplicación se ha estructurado en tres escenas para garantizar un uso sencillo e intuitivo: la pantalla de inicio, la interfaz principal y el menú de opciones. A continuación, se examinará la composición de cada una de ellas, detallando los elementos visuales e interactivos que las integran. Asimismo, se expondrá el código en C# desarrollado para dotar de funcionalidad a la aplicación y gestionar las transiciones entre los distintos apartados.

3.4.1 Escena principal

La escena principal es centro de la aplicación, ya que es el espacio donde se desarrolla el funcionamiento de esta. En este apartado detallaremos los componentes que conforman su estructura, desde los elementos de la interfaz de usuario (UI) hasta los objetos interactivos presentes en la pantalla. Se procederá a detallar detenidamente los scripts asociados a esta escena, explicando cómo el código se encarga de procesar las imágenes captadas por la cámara mediante YOLO. La escena (Figura 3-6) está compuesta por tres objetos principales como son el *Box container*, GestorIA y el canvas el cuál es el elemento que contiene todos los elementos de la Interfaz de Usuario.

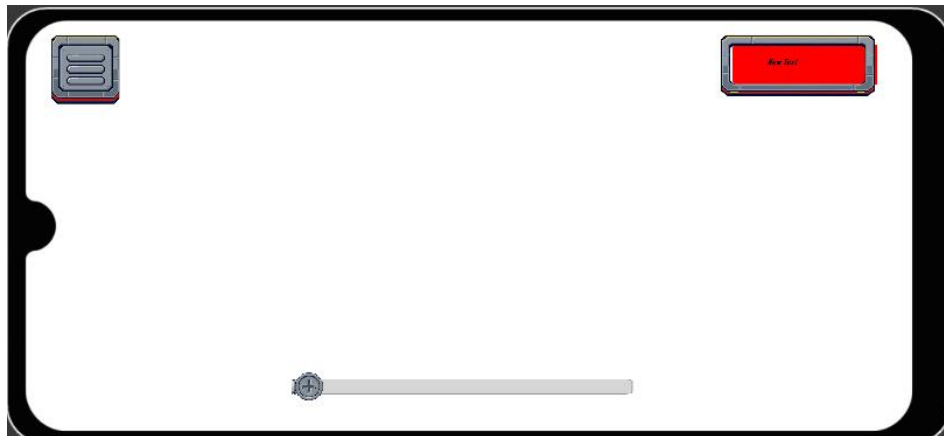


Figura 3-6 Escena principal (Elaboración propia)

A continuación, se describen los elementos del canvas:

1. *FadeOverlay*: Creamos una imagen en color negro y hacemos que ocupe la totalidad de la escena, este elemento permite un difuminado entre la escena de comienzo y la propia escena principal (Figura 3-7).

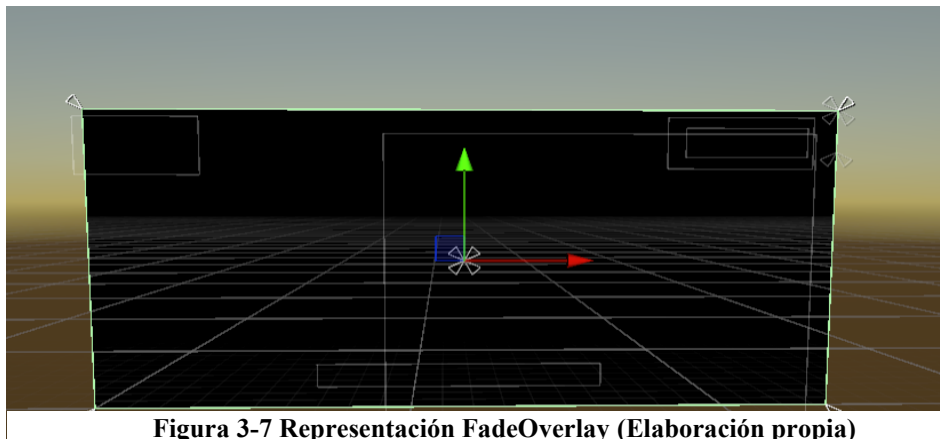


Figura 3-7 Representación FadeOverlay (Elaboración propia)

2. *WebcamScreen*: Creamos una imagen en color negro y hacemos que ocupe la totalidad de la escena este elemento es el lienzo donde se representará el video dado por la cámara (Figura 3-8).

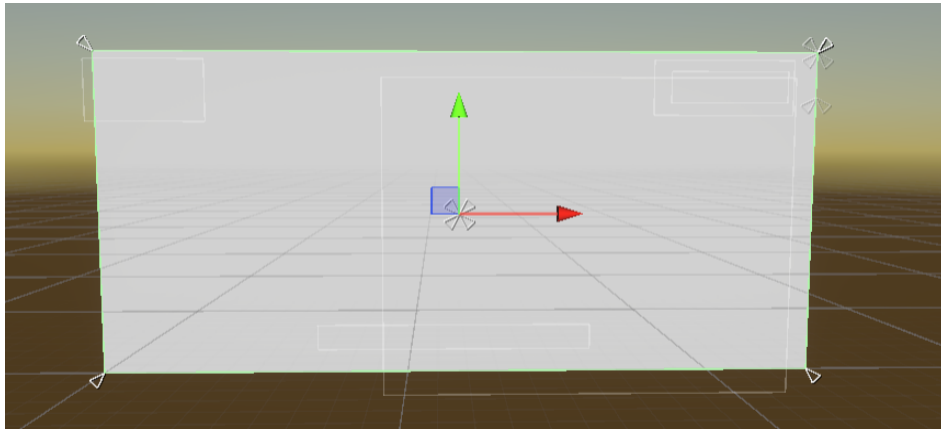


Figura 3-8 Representación WebcamScreen (Elaboración propia)

3. *TextBackground*: Creamos una imagen en color rojo y ajustamos hasta que este con un tamaño adecuado. Se trata del fondo del recuadro donde aparece si se ha detectado un barco, este cambia de color entre rojo y verde dependiendo del estado de detección (Figura 3-9).

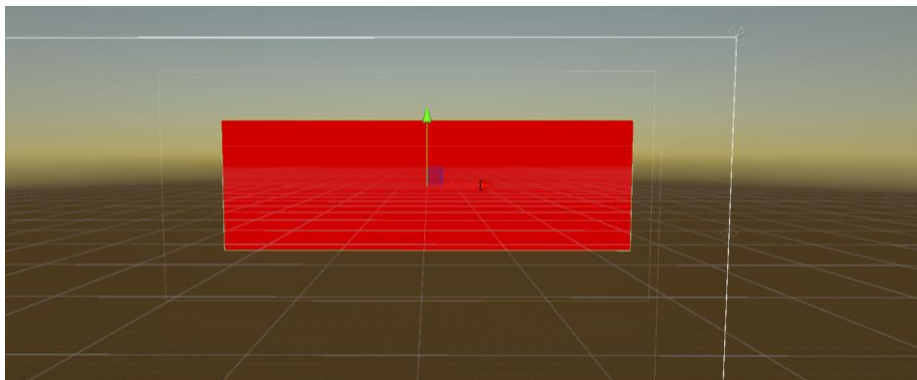


Figura 3-9 Representación TextBackground (Elaboración propia)

3.1/2 *DetectionPanel/Label*: Se procede a crear una etiqueta y un panel, posteriormente es desplazado a la esquina superior derecha de la pantalla ajustándolos junto con *TextBackground* es decir la etiqueta, después se procede a cargar la imagen del *Detection panel* como se ve en la Figura 3-10.



Figura 3-10 Representación Detection Panel (Elaboración propia)

4. **BotonAjustes:** Se crea un botón, a continuación, se desplaza a la esquina superior izquierda de la pantalla alineando con *TextBackground*, cuando ya se tiene alineado se procede a cargar la imagen del BotonAjustes como se ve en la Figura 3-11 este será el encargado de cambiar de escena hasta el menú de ajustes.

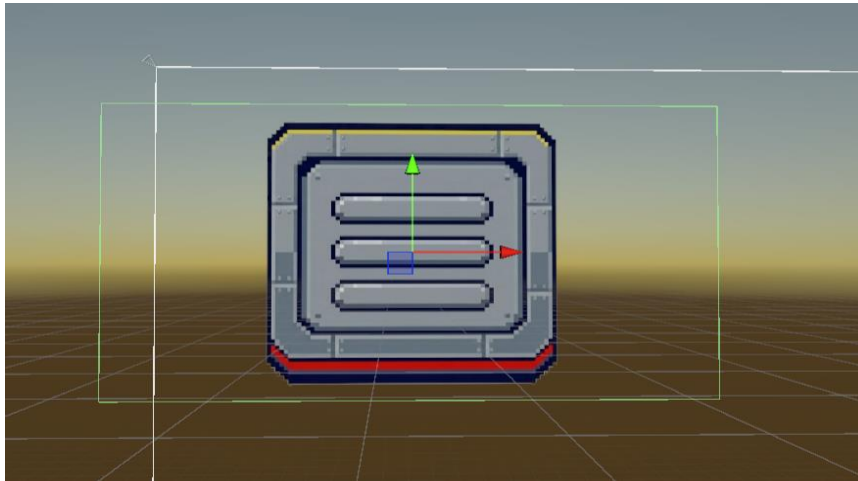


Figura 3-11 Representación Botón ajustes (Elaboración propia)

5. **ZoomSlider:** Se crea un *slider* y desplaza a la parte de debajo de la pantalla, cuando ya se tiene alineado se procede a cargar la imagen del ZoomSlider como se ve en la Figura 3-12 este será el encargado de realizar *zoom* de la cámara en la escena.

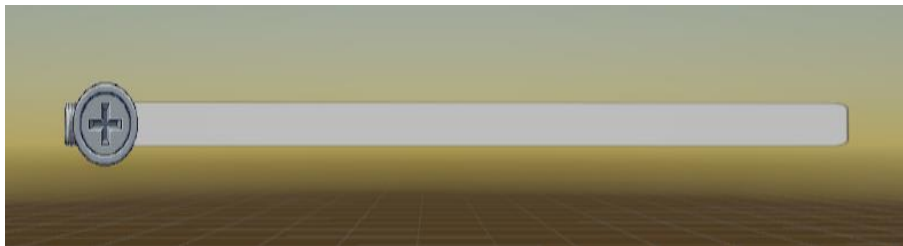


Figura 3-12 Representación ZoomSlider (Elaboración propia)

El siguiente objeto en la jerarquía, ya fuera del Canvas es el GestorIA. Este objeto vacío alojará el código YoloDetector.cs para incorporarlo a la escena. Al seleccionarlo se puede ver en el inspector una serie de opciones que son las variables públicas de configuración del código (Figura 3-13).

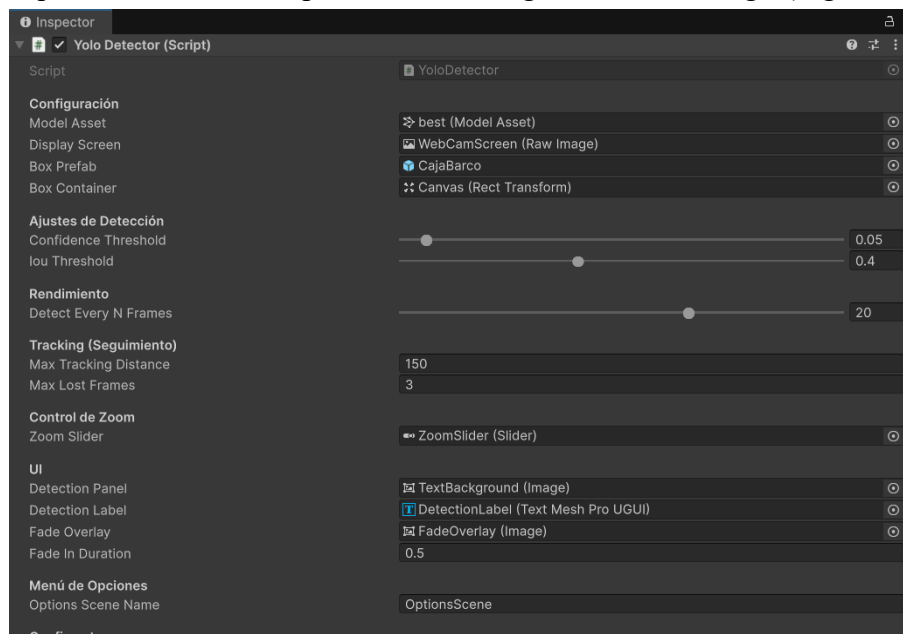


Figura 3-13 Panel variables YoloDetector (Elaboración propia)

Todos los parámetros se tratan de variables públicas que se encuentran definidas en el código de YoloDetector.cv (Anexo I, líneas 1 a la 66) y se integran en diferentes secciones como son:

- Configuración: Incluye la selección del modelo de YOLO preentrenado de entre las dos variantes (entrenado con el dataset de COCO o con el de SeaShips) almacenados en formato onnx en la sección *Model Asset*, el *Display Screen* que se refiere al canvas donde se recoge los *frames* de la cámara, *Box Prefab*, u objeto modelo con el recuadro donde se enmarcaría el barco detectado y el *Box container* para contener los objetos instanciados.
- Ajustes de Detección: Los ajustes de detección en los que se encuentra la confianza y el verificador de detección
- Rendimiento: El intervalo de detección del sistema, detectando a partir de un número específico de *frames*.
- Seguimiento: Distancia máxima de píxeles que puede haber para que no se realice seguimiento y el número de pérdida de *frames* por el cual si la diferencia es superior se clasificaría como otro contacto.
- Control de *zoom*: Para realizar *zoom* a la imagen
- UI: Se configuran los elementos de la jerarquía correspondientes al visor de detección, de manera que la interfaz cambie de color dinámicamente y muestre el conteo actualizado de barcos en pantalla.
- Menú de Opciones: Se define la escena de destino que el sistema debe cargar cuando el usuario presiona el botón de opciones.
- Config externa: Importa un archivo de ajustes globales al iniciar la aplicación, el cual establece los valores estándar y los parámetros predeterminados para el correcto funcionamiento del modelo.

A continuación, se procederá a explicar el código por secciones para un mayor entendimiento de su funcionamiento, empezando por el método *Start ()*, que en Unity se ejecuta en el primer frame:

1. Configuración de la Interfaz y Pantalla (líneas 92 a 113)

El script en las líneas 92 a 113 comienza preparando el entorno visual. Pone el panel de superposición (*FadeOverlay*) completamente en negro y bloquea la orientación de la pantalla del dispositivo para que se mantenga en horizontal. Además, también inicializa el slider del *zoom* para que empiece en el nivel mínimo.

2. Estabilización del renderizado (líneas 116 a 117)

Mediante este script, se indica al motor gráfico que espere exactamente dos fotogramas completos. Esto garantiza que todos los elementos visuales de la interfaz se hayan acomodado correctamente en la pantalla antes de continuar.

3. Solicitud de Permisos en Android (líneas 119 a 131)

Si la aplicación se compila para un dispositivo Android, verifica si el usuario ha otorgado permiso de uso de la cámara. Si no tiene autorización, invoca a la interfaz nativa del teléfono mediante *Permission.RequestUserPermission*. La corrutina se pausa (*WaitUntil*) hasta que el usuario tome una decisión (aceptar o rechazar).

4. Carga del Modelo de Inteligencia Artificial (líneas 133 a 134)

Carga el archivo del modelo YOLO (asignado en la variable *ModelAsset*) en la memoria. A continuación, crea un *worker* (trabajador) que es el objeto responsable de ejecutar la red neuronal, configurada aquí para usar el procesador del dispositivo (*BackendType.CPU*).

5. Inicialización de la Cámara Web (líneas 137 a 154)

El script obtiene una lista de todas las cámaras del dispositivo. Selecciona prioritariamente la cámara trasera (es decir, la que no es *isFrontFacing*), se configura una textura *WebCamTexture* a 1280x720 y

se empieza a capturar la imagen con el método `Play ()`. Se espera un momento hasta estar seguro de que la cámara esté mandando imágenes reales (`webcam.width > 16`) antes de ajustar la proporción del visualizador gráfico (`AdjustDisplay ()`).

6. Preparación de Tensores de YOLO (líneas 156 a 167)

Se actualiza el texto en la interfaz que indica el número de barcos detectados (poniéndolo en 0 de inicio). Después crea la *RenderTexture* (una textura en blanco de 640x640) y los tensores de entrada (las estructuras de matrices matemáticas que procesará el *worker*).

7. Importación de Configuraciones Finales y Animación (líneas 168 a 182)

Una vez que todas las variables están instanciadas, activa `isReady = true` para que el método `Update ()` pueda empezar a trabajar. Si hay una configuración global guardada (por ejemplo, desde el menú de opciones), sobrescribe los ajustes por defecto. Finalmente, inicia otra pequeña rutina (*FadeInFromBlack*) que vuelve gradualmente transparente la pantalla negra inicial.

Proseguiremos ahora con el método `Update ()`, que en Unity se ejecuta cada *frame*, y está subdividido en dos tareas principales (líneas 247 a 293):

1. Leer resultados pendientes: Si el programa terminó de analizar un fotograma (*IsReadbackRequestDone*), extrae los datos del procesador (*ReadbackAndClone*) y los envía a procesar.
2. Enviar un nuevo frame: Usa un contador (*frameCounter*). Si se alcanza el número de fotogramas esperado (*detectEveryNFrames*), recorta la imagen de la cámara según el nivel de *zoom*, la transfiere a una *RenderTexture*, la convierte en un tensor matemático y le ordena al *worker* de YOLO que empiece a analizarla.

Cuando la red neuronal devuelve un resultado, lo hace en forma de matriz numérica multidimensional (tensor). El programa (líneas 326 a 367) recorre estas predicciones para buscar qué tipo de barco es (buscando el puntaje o score más alto entre las clases) y comprueba si supera el umbral de confianza (*confidenceThreshold*). Como YOLO suele detectar el mismo objeto varias veces superponiendo cajas, se utiliza la función NMS (*Non-Maximum Suppression*) junto con *CalculateIoU* para eliminar las cajas duplicadas que se solapan demasiado (*iouThreshold*).

Para que las cajas no sean intermitentes y tengan un número identificador asignado usamos (*UpdateTracking*), este método (líneas 369 a 423) compara las nuevas detecciones con las del fotograma anterior usando la distancia matemática (*Vector2.Distance*). Si una detección nueva está cerca de donde estaba un barco antes, asume que es el mismo barco. Si un barco no se detecta durante unos pocos fotogramas (*maxLostFrames*), se oculta temporalmente antes de eliminarlo por completo.

Para graficar los recuadrados sobre los barcos, el script (líneas 441 a 502) calcula la escala entre el tamaño original de YOLO que son 640 píxeles y el tamaño real de la pantalla. Para no saturar la memoria creando y destruyendo objetos visuales (*Instantiate* y *Destroy*), utiliza un patrón llamado *Object Pool* que tiene una lista de cajas (*boxPool*) que simplemente apaga o enciende (*SetActive*) según las vaya necesitando.

Continuando con el control del *zoom* (líneas 301 a 324) usando *AdjustDisplay* estira o recorta el componente de interfaz *RawImage* asegurándose de que la proporción de la imagen no se deforme (evita que la cámara se vea aplastada). *AplicarZoomVisual* altera las coordenadas UV de la textura de la cámara; al reducir el área visible en pantalla, crea un efecto de *zoom* digital.

Para finalizar el código termina con un bloque necesario (líneas 514 a 529) para aplicaciones de IA o con cámaras. Cuando se cierra la escena o se destruye el objeto, *OnDestroy* apaga la cámara y vacía explícitamente (*Dispose* o *Release*) los tensores, texturas y el modelo de YOLO. Si esto no se hiciera, la memoria RAM se llenaría y la aplicación colapsaría. Además del módulo de cambio de escena para ir al menú de opciones.

3.4.2 Menú de opciones

El menú de opciones es centro de configuración de parámetros de la aplicación en tiempo real, ya que es el espacio donde se cambian una serie de parámetros preestablecidos para ajustar su funcionamiento. En este apartado detallaremos los componentes que conforman el menú y el *script* asociado a esta escena, explicando cómo el código se encarga de sobrescribir los parámetros e implementarlos al volver a la escena principal.

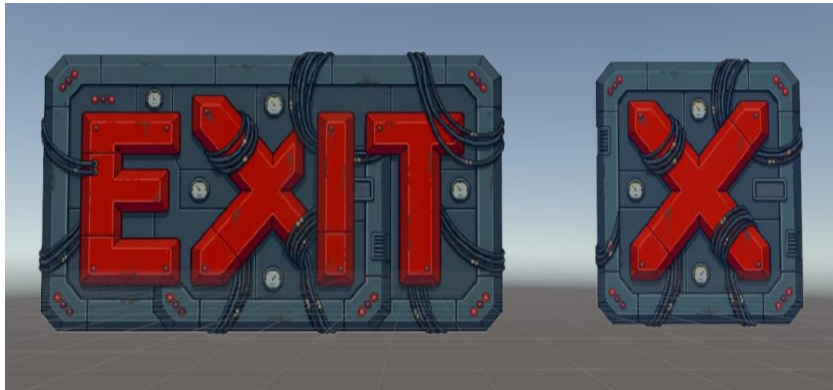


Figura 3-14 Representación botones de cerrar y salida (Elaboración propia)

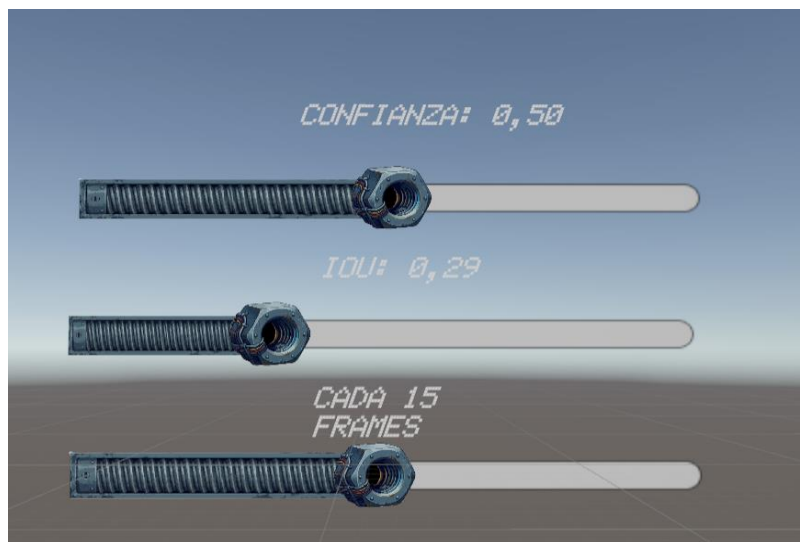


Figura 3-15 Representación Sliders (Elaboración propia)

En el menú se pueden visualizar tres *sliders* (Figura 3-15) y dos botones (Figura 3-14). Estos sliders sirven para modificar tres parámetros clave que son: el umbral de confianza, que determina la probabilidad mínima que tiene que tener un objeto para poder ser considerado un barco; el IOU que sirve para evitar el solapamiento y duplicación de las cajas de modo que la escena principal este más depurada.; y el intervalo de *frames*, que sirve para que el sistema realice una inferencia cada número de *frames* seleccionado, lo que flexibilízala demanda de trabajo exigible al dispositivo. Por otra parte, hay dos botones una de salida de la aplicación y otro para cerrar el menú y volver a la escena principal.

En referente a la composición de la escena en este caso, vuelve a usarse *FadeOverlay* para conseguir ese difuminado negro entre escenas. En contraparte con la escena principal, se compone de tres imágenes superpuestas. Un fondo *pixelart* de un pasillo de un buque, un rectángulo azul semitransparente y un marco de misma estética. Dentro del marco se encuentran los *sliders* previamente mencionados y el botón de salida mientras el botón de cierre del menú se dispone en la esquina superior derecha (Figura 3-16).

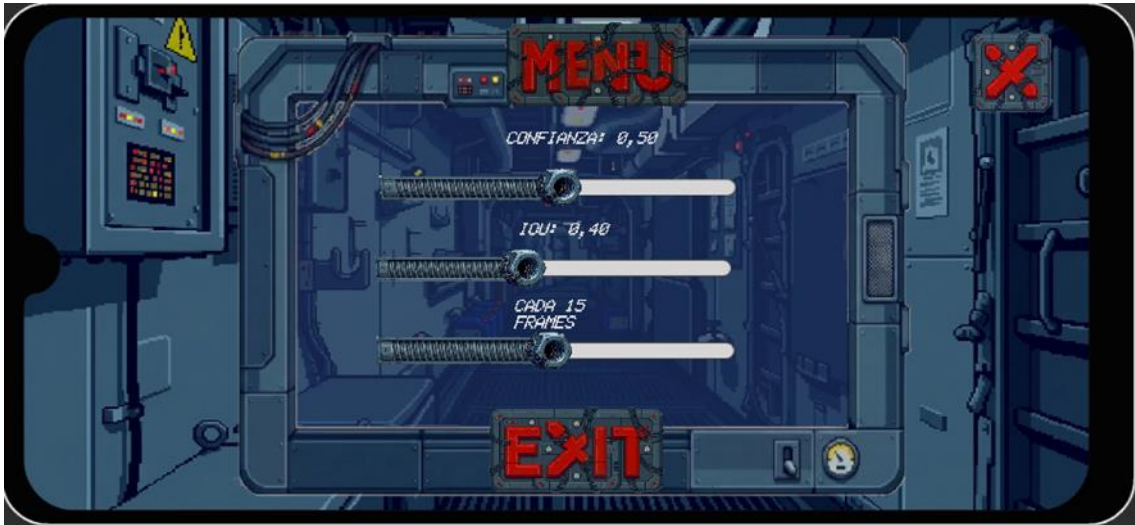


Figura 3-16 Escena menú (Elaboración propia)

El *script* `optionmenucontroller.cs` que maneja la escena, recogido en el Anexo II, se encuentra contenido en un objeto denominado *optionscontroller* (véase Figura 3-17), donde se asignan los *sliders* a sus variables y se cambian los textos en función de los parámetros en tiempo real. Además, en el apartado de escena se puede seleccionar a que escena se debe redirigir cuando se pulsa el botón de cerrar. Entrando en la explicación del código, en primera instancia se importan las librerías necesarias y se declaran las variables de los *sliders* y los botones (líneas 01 a 25).

El código continúa con el bloque `Start ()` (líneas 26 a 88) que se ejecuta una sola vez justo cuando entras al menú de opciones realizando las siguientes acciones:

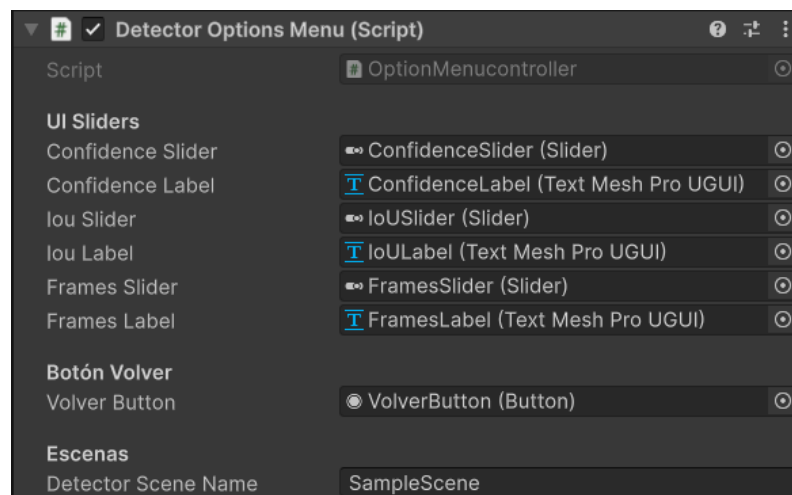


Figura 3-17 Panel variables DetectorOptiosMenu(Elaboración propia)

- Conexión con la configuración previa: Primero busca *DetectorConfig.Instance*, si no lo encuentra como, por ejemplo, si se ejecuta la escena de opciones directamente sin pasar por el menú inicial, lanza un error en rojo *Debug.LogError* y detiene la ejecución *return* para que el juego no colapse.
- Límites de los *sliders*: Se configuran los límites numéricos de los sliders para garantizar que los valores introducidos se mantengan dentro de un rango válido. Por ejemplo, *confidence* y *iou* van de 0 a 1 es decir 0% a 100%. Los *frames* van de 1 a 30, y se usa la función *wholeNumbers = true* para que no se pueda elegir "2.5 frames", sino exclusivamente números enteros.

- Carga de valores previos: Asigna a los sliders los valores que ya estaban guardados en el detector para que no empiece desde cero. Finalmente, llama a *UpdateLabels ()* para actualizar el texto en pantalla.

Dentro del método *Start ()*, se inicializa la interactividad de la interfaz. Utilizando el método *onValueChanged.AddListener*, el sistema queda a la espera de las interacciones; de este modo, cada vez que se modifica el valor numérico de un *slider*, dicho valor se almacena automáticamente en el archivo de configuración (*config*) y se invoca la función *UpdateLabels ()* para refrescar la información mostrada en pantalla. En el caso específico del *slider* correspondiente a los *frames*, conversión explícita de tipo (*int*) para asegurar que el dato devuelto se registre como un número entero. Por último, se configura el comportamiento del botón de retorno: mediante código se vincula el clic (*onClick*) al método *VolverAlDetector ()*, que se encarga de gestionar la transición hacia la escena principal.

Los métodos auxiliares son tres:

1. *UpdateLabels ()*: Toma el valor numérico y lo convierte en texto para la pantalla. El: F2 es una función de formato que trunca los decimales y muestra solo 2.
2. *VolverAlDetector ()*: Carga la escena principal con el gestor de escenas de Unity, usando la escena preseleccionada en la variable *detectorSceneName*.
3. *SalirDeLaApp ()*: Al ejecutarse en Android o PC, la instrucción *Application.Quit ()* cierra la aplicación por completo. En el editor de Unity esto no hace nada visible, por lo que se añade un *Debug.Log* para confirmar que la acción funciona.

3.4.3 Escena inicial

En este apartado se describirá la escena de inicio que aparece inmediatamente después de inicializar la aplicación, sirve de preámbulo y precarga para la escena principal e introduce al usuario en un entorno con ambientación naval.

En la composición de la escena mostrada en la Figura 3-18, se puede visualizar un fondo *pixelart* inspirado en la Escuela Naval Militar. En la esquina superior derecha se encuentra un título en el mismo estilo artístico con el texto *Ship detector*. En la esquina superior izquierda está dispuesto el botón de mute para apagar la música de fondo que suena al inicializar la aplicación. En el centro de la pantalla superponiéndose el fondo se encuentra el botón de inicio para acceder a la escena principal.



Figura 3-18 Escena de inicio (Elaboración propia)

El *script startscreencontroller.cs*, referenciado en el Anexo III, que controla esta escena está contenido en el objeto *Startconfigcontroller*. Además, para integrar la música de fondo[54] en la escena, se ha añadido un componente *Audio Source* manteniendo sus parámetros por defecto. Únicamente se ha modificado la opción de reproducción continua (*Loop*), activándola desde el inspector, tal y como se ilustra en la Figura 3-19.

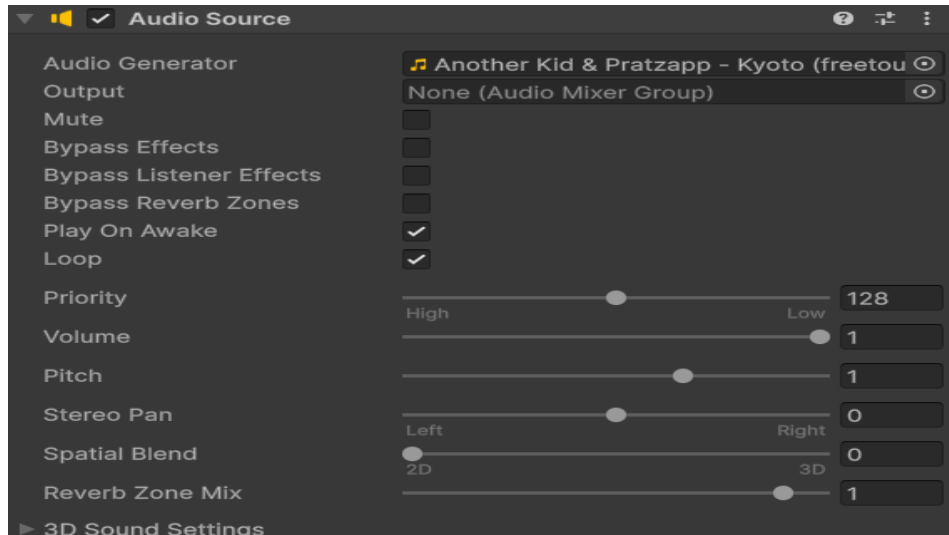


Figura 3-19 Audio Source Panel (Elaboración propia)

Entrando en la explicación del código, en primera instancia se importan las librerías necesarias y se declaran las variables de los botones. Después se utiliza las funciones: (*isLoading=false*) que evita que se pulse *Start* varias veces y (*isAudioOn=true*) que guarda si el sonido está activado o no.

Al iniciar la escena, el método *Start ()* (líneas 28 a 47) hace cuatro cosas en orden: Fuerza la orientación horizontal, pone el *overlay* negro completamente invisible, conecta cada botón con su función correspondiente y muestra el icono correcto del botón de audio.

Cada vez que el usuario pulsa el botón de audio (líneas 49 a 65), *ToggleAudio ()* invierte el estado (de activado a desactivado o viceversa) y silencia o activa la música. Luego *UpdateAudioIcon ()* cambia el icono del botón para reflejar visualmente ese estado.

Por otra parte, tenemos la configuración del botón de *Start* (líneas 67 a 79). Cuando el usuario pulsa el botón, este método comprueba que no se haya pulsado ya antes, oculta los botones de pantalla, activa el *overlay* para bloquear más toques durante la transición y activa la corrutina de fundido.

Para finalizar (líneas 81 a 112), esta corrutina ejecuta *frame a frame*. Durante el tiempo definido en *fadeOutDuration*, la pantalla se va volviendo negra progresivamente, el volumen de la música baja hasta 0. Al finalizar, espera 0,2 segundos para asegurar que todo está negro y carga la escena del detector de forma asíncrona evitando congelar la aplicación.

3.5 Entrenamiento con dataset específico.

Aunque la versión inicial del modelo utilizaba el *dataset* genérico COCO, su capacidad resultaba limitada para las necesidades de este proyecto. Para resolver esta carencia y dotar a la red neuronal de la capacidad de clasificar distintos buques comerciales y pesqueros, se ha entrenado un modelo propio con el *dataset* SeaShips, justificado por su extenso volumen de imágenes en entornos reales y su categorización en seis clases de buques. Por consiguiente, en el presente apartado se procederá a exponer el proceso de entrenamiento de un YOLO11 en su versión nano utilizando dicho conjunto de datos. Para el entrenamiento de este sistema se ha hecho uso de un servidor del Centro Universitario de la Defensa

de Marín, el cual ha sido usado para el entrenamiento de la IA mediante GPU. La tarjeta gráfica para usar fue una NVIDIA GeForce RTX 3090 Ti.

El primer paso consistió en obtener los datos de entrenamiento. Para ello, se recurrió a Roboflow, una plataforma web especializada en el alojamiento, gestión y preprocesamiento de *datasets* para modelos de visión artificial. Desde dicho repositorio no se descargó el *dataset* SeaShips en su totalidad, sino un Seaships 7000 [55], un subconjunto de 7000 imágenes ampliamente utilizado en el campo de la visión artificial para el entrenamiento de modelo de detección de barcos en tiempo. Estos datos se descargaron ya estructurados para la arquitectura YOLO11; esto implica que las coordenadas de las cajas delimitadoras están normalizadas según el estándar de la versión de YOLO y que el conjunto incluye el archivo de configuración `-yaml` necesario para estructurar las rutas de entrenamiento y validación, así como las etiquetas de las clases. Posteriormente, se inicializó el entorno de trabajo mediante Jupyter Lab, el cual proporciona una interfaz de desarrollo para ejecutar código en el servidor. A través de este entorno, se lanzaron los comandos e instrucciones encargados de llevar a cabo el entrenamiento aprovechando la GPU del sistema.

En primera instancia se realiza el comando `%pip install ultralytics`, para descarga el paquete oficial desde el índice de paquetes de Python para que esté disponible de inmediato en el entorno de desarrollo. El paquete *ultralytics* proporciona acceso a la arquitectura YOLO y brinda las herramientas necesarias para entrenar y ejecutar modelos. Cabe reseñar que esta biblioteca requiere un entorno configurado con Python 3.8 o una versión más reciente para funcionar correctamente. Posteriormente subimos a Jupyter el archivo SeaShips.zip y lo extraemos en el servidor.

Después se procede con el entrenamiento de mismo usando un bloque de código (líneas 01 a 33), dividido en cuatro partes:

1. Navegación al directorio del *dataset*: Las primeras líneas utilizan las librerías estándar de Python `os` y `pathlib` para entrar en la carpeta donde se descargaron y descomprimieron previamente los datos (SeaShips.v1-seaships.yolov11).
 - Se utiliza `os.chdir()` para cambiar el directorio de trabajo activo de Python hacia esa carpeta, lo cual es fundamental para que el resto de los comandos encuentren los archivos correctos.
 - `Print` confirma en pantalla la nueva ubicación para asegurarse de que el paso anterior funcionó.
2. Verificación de los datos: Dentro del directorio, el código ejecuta comandos de la terminal de Linux para auditar el estado del *dataset*.
 - `!ls -la`: Muestra todos los archivos y carpetas. Ultralytics espera que las imágenes y etiquetas estén divididas típicamente en `train/`, `valid/` y opcionalmente `test/`.
 - `!ls train/images/ | wc -l`: Cuenta exactamente cuántas imágenes hay en la carpeta de entrenamiento.
 - `!head -5 data.yaml`: Imprime las primeras 5 líneas del archivo de configuración principal (YAML), el cual contiene las rutas a las imágenes y la lista de clases a detectar (tipos de barcos).
3. Corrección de rutas en el archivo YAML: Los *datasets* descargados a menudo traen rutas absolutas o relativas rotas que apuntan a carpetas superiores (`../`). Si Ultralytics intenta leer el YAML y no encuentra las imágenes, el entrenamiento fallará.
 - El *script* abre `data.yaml` en modo lectura y carga su contenido.
 - Utiliza el método `replace()` para quitar el `../` de las rutas de entrenamiento, validación y test, cambiándolas por rutas relativas limpias que apuntan a la carpeta actual.
 - Guarda el resultado en un nuevo archivo seguro llamado `data_fixed.yaml`, manteniendo el original intacto por si hay errores.
4. Entrenamiento del modelo YOLO11: Se llama a la biblioteca Ultralytics instalada previamente para iniciar el entrenamiento.

- YOLO ('yolo11n.pt'): Carga los pesos preentrenados del modelo "Nano" de YOLO11. Al ser la versión más ligera (la "n"), es muy rápida y consume pocos recursos.
- data='data_fixed.yaml': Le indica al modelo que utilice el archivo de configuración corregido en el paso 3 para encontrar las imágenes y sus etiquetas.
- epochs=100: El modelo verá el *dataset* completo 100 veces como límite para aprender a detectar barcos.
- imgsz=640: Redimensiona todas las imágenes a 640x640 píxeles antes de pasarlas a la red neuronal.
- device=0 y batch=32: Le dice a Ultralytics que use la tarjeta gráfica principal (GPU 0, RTX 3090 Ti de 24GB) y que procese 32 imágenes a la vez en cada iteración de aprendizaje.
- patience=20: Es un mecanismo de "Early Stopping" (parada temprana). Si el modelo no mejora su precisión en 20 épocas consecutivas, el entrenamiento se cancelará automáticamente para ahorrar tiempo y evitar el sobreajuste.

El entrenamiento tuvo una duración de 38 minutos y 100 épocas. Una vez completado, el modelo entrenado de tipo Pytorch se transforma al formato ONNX (*Open Neural Network Exchange*) para poder ser usado en la aplicación desarrollada (líneas 34 a 49).

Al finalizar y con el ONNX ya creado se procede a realizar un test de comprobación para verificar que es plenamente funcional seleccionando una imagen de la carpeta de validación del *dataset* de forma aleatoria y dando instrucciones de que diga si encontró un buque el tipo y la probabilidad (líneas 50 a 80).

Posteriormente se realiza la instalación del paquete Pandas con el comando `%pip install pandas`, la biblioteca de código de Python para la manipulación y el análisis de datos, desarrollada en 2008 y cuyo nombre proviene del término econométrico *Panel Data* (datos de panel). Funciona, en términos prácticos, como el equivalente a Microsoft Excel dentro del ecosistema de programación de Python. Se utilizará para realizar una serie de gráficas para ejemplarizar la mejora de detección con respecto al modelo base pre-entrenado con COCO.

Finalmente, para mostrar cómo ha mejorado el sistema de detección, se ha sometido a los dos modelos a una prueba simultánea en la que se les han brindado 30 imágenes aleatorias del set de validación de SeaShips, graficando una imagen comparativa entre ambas (líneas 81 a 127).

Como resultado, se obtiene una imagen, tal y como se puede observar en la Figura 3-20.

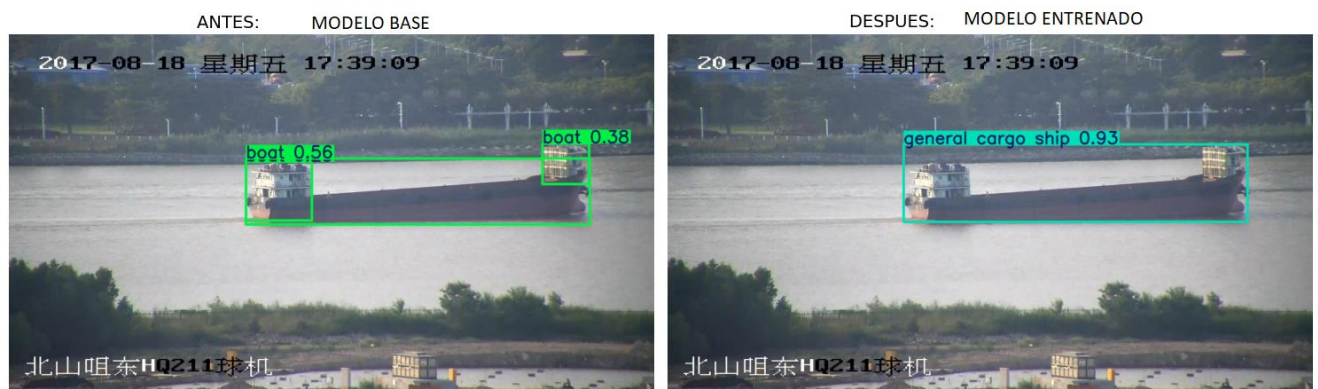


Figura 3-20 Comparativa de modelos (Elaboración propia)

4 VALIDACIÓN Y PRUEBAS

En este capítulo se presentarán los resultados finales de este proyecto, junto con las métricas de evaluación empleadas. Asimismo, se incluirán las pruebas realizadas en entornos reales para validar su funcionamiento.

4.1 Resultados de los modelos (COCO vs propio)

En el presente apartado se dispondrá a realizar una serie de pruebas para realizar una valoración y validación de la aplicación desarrollada en cuestión. El código utilizado en estas pruebas se recoge en el Anexo VI. Estas comparativas se realizaron con dos versiones de YOLO 11 versión nano, una con pre-entrenada con COCO y otra con un entrenamiento propio usando SeaShips 7000.

4.1.1 Comparativa (conjunto de validación SeaShips)

A continuación, se llevarán a cabo diversas pruebas utilizando el conjunto de imágenes de validación del subconjunto de SeaShips empleado. Es importante señalar que dicho subconjunto no se ha utilizado durante el entrenamiento del modelo, sino exclusivamente para su fase de validación.

En primer lugar, se selecciona el conjunto de validación, se contabiliza el número de barcos reales a partir de las anotaciones disponibles. Posteriormente, se cargan ambos modelos y, empleando un umbral de confianza del 65%, se procesan las imágenes en lotes de 32, realizando la detección sobre

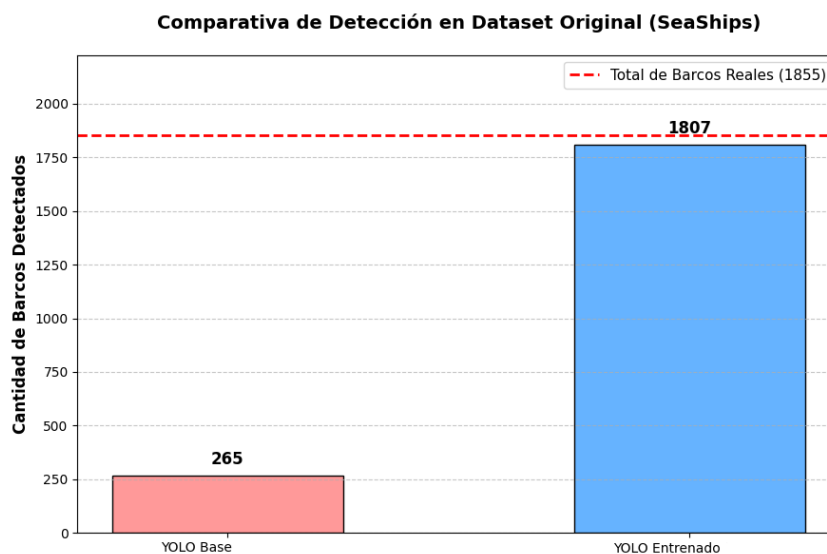


Figura 4-1 Grafica comparación (Elaboración propia)

cada bloque y acumulando el número total de predicciones generadas por cada modelo que arroja esta grafica Figura 4-1.

En el conjunto de datos de SeaShips se registran 1855 barcos reales según las anotaciones. El modelo YOLO11n base detecta únicamente 265 embarcaciones, evidenciando un bajo *recall* debido a su entrenamiento generalista en COCO. Por el contrario, el modelo entrenado específicamente identifica 1807 barcos, lo que demuestra una capacidad de detección notablemente superior y un ajuste efectivo al dominio del entorno.

Posteriormente, se ha realizado una comparativa más exhaustiva calculando métricas, tales como la precisión (proporción de detecciones correctas entre todas las positivas), el *recall* (proporción de positivos reales que son detectados) y la *mAP50* (precisión media promedio a un umbral de *IoU* del 50%), que arroja los resultados mostrados en la Figura 4-2.

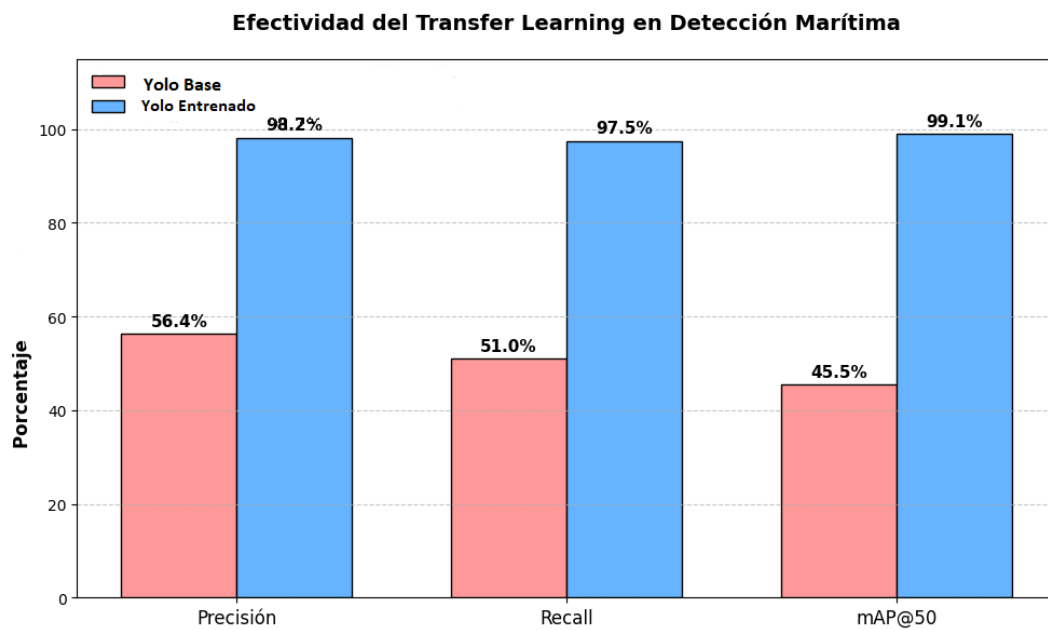


Figura 4-2 Métricas de clasificación (Elaboración propia)

El modelo base alcanza una precisión del 56,4%, un *recall* del 51% y un *mAP50* del 45,5%, mientras que el modelo entrenado logra valores superiores: 98,2% de precisión, 97,5% de *recall* y 99,1% de *mAP50*. Estos resultados demuestran mejoras notables en todas las métricas, lo que valida su efectividad para la detección marítima.

4.1.2 Comparativa de detección visual

En este apartado se expone una serie de imágenes comparativas para mostrar la mejora de detección del sistema. Cabe destacar que las imágenes utilizadas para estas comparaciones pertenecen al conjunto de validación de SeaShips [55].

Como se puede observar en la Figura 4-3 el modelo entrenado realiza una clasificación más específica y con mayor porcentaje de confianza que el modelo base, además de encontrarse la totalidad del buque en el marcador en el modelo entrenado, en contraparte del modelo base que solo recuadra una sección del buque.



Figura 4-3 Comparativa detección N°1 (Elaboración propia)

En la Figura 4-4 se puede observar que el modelo base es incapaz de detectar los dos buques mientras que en el modelo entrenado ambos están detectados y clasificados adecuadamente mostrando la robustez del modelo.



Figura 4-4 Comparativa detección N°2 (Elaboración propia)

En la Figura 4-5 ambos modelos detectan la embarcación, pero cabe destacar que el modelo entrenado da una clasificación precisa y encuadra la totalidad del buque en su marcador.

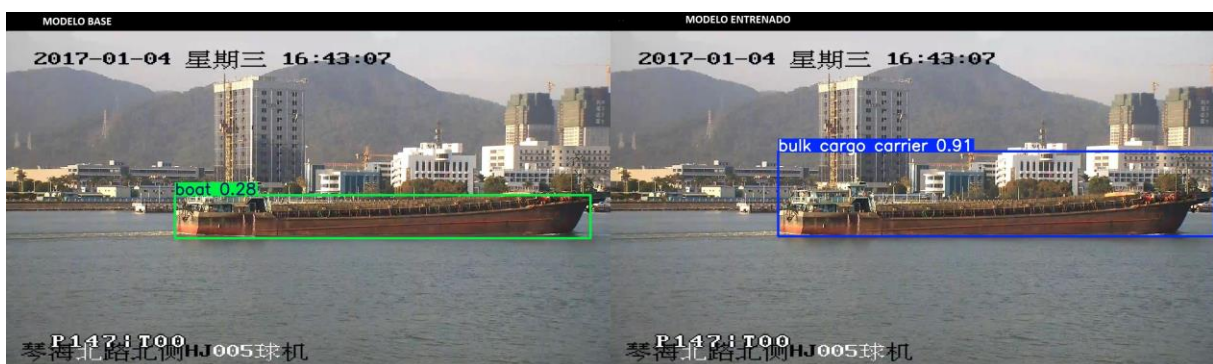


Figura 4-5 Comparativa detección N°3 (Elaboración propia)

Como se puede visualizar en Figura 4-6, el modelo base detecta un falso positivo confundiendo un objeto con un barco. Por otra parte, aunque ambos modelos detectan las dos embarcaciones, el modelo entrenado reconoce toda la eslora de ambos buques y vuelve a realizar un mejor encuadre.



Figura 4-6 Comparativa detección N°4 (Elaboración propia)

Como se puede observar, el modelo entrenado es claramente superior, en cuanto a la detección de las embarcaciones y el encuadre, a diferencia del modelo base que en varias ocasiones genera falsos positivos o no llega a detectar ninguna embarcación.

4.2 Prueba de la aplicación en entorno real

En este apartado se exponen una serie de capturas de la aplicación durante las pruebas realizadas en un entorno marítimo real para su visualización y validación.

La primera prueba se realiza durante una salida en la Lancha de Instrucción GM Barrutia en la Ria de Pontevedra. Inicialmente, se prueba el sistema con embarcaciones atracadas, buscando validar la detección, aunque presenten estructuras o edificaciones detrás suya. Como se puede observar en la Figura 4-7, se obtiene un resultado positivo detectando dos de las tres embarcaciones.



Figura 4-7 Captura “Saliendo de la dársena” (Elaboración propia)

Escasos minutos más tarde se realiza otra prueba al situarse GM Barrutia en la salida de la dársena hacia la ría. Como se aprecia en la Figura 4-8, la aplicación detecta dos veleros por la proa y un pequeño pesquero, mientras mantiene el contacto del Tabarca. Esto indica que el sistema es capaz de detectar varios tipos de barcos de diferentes tamaños y clases. Por otra parte, también nos sirve para comprobar que se mantienen los contactos hasta perder del todo el buque, como es el caso del Tabarca en este caso.



Figura 4-8“Entre puntas” (Elaboración propia)

Para finalizar se realiza una última verificación con varios veleros, validando la funcionalidad del *tracking* con sus cambios de rumbo. Se comprueba también la correcta detección de estos a pesar de su pequeño tamaño (Figura 4-9).



Figura 4-9“Ría de Pontevedra” (Elaboración propia)

5 CONCLUSIONES Y LÍNEAS FUTURAS

5.1 Conclusiones y revisión de los objetivos

En referente a los objetivos iniciales del proyecto, el objetivo principal ha sido superado. Se ha desarrollado una aplicación móvil que mediante visión artificial detecta embarcaciones en tiempo real, mostrando un seguimiento con marcadores sobre las mismas.

En cuanto a la consecución de los objetivos específicos iniciales, en primer lugar, se realizó un estudio del estado del arte. Esto permitió analizar las diferentes arquitecturas de detección de objetos disponibles y seleccionar el modelo que mejor se adaptaba a los requisitos del proyecto.

Posteriormente, se llevó a cabo el desarrollo inicial de la aplicación utilizando un modelo Base de YOLO11n preentrenado con COCO. Esta fase sirvió para establecer la estructura base del sistema y comprobar la viabilidad de la detección de imágenes antes de enfocar el proyecto hacia los siguientes objetivos.

Para superar las limitaciones de dicho modelo, se procedió al entrenamiento de un modelo propio utilizando el subconjunto de *dataset* SeaShips7000, lo que mejoró notablemente su rendimiento en el ámbito marítimo. Asimismo, se desarrolló una interfaz de visualización intuitiva diseñada para que el usuario pueda interactuar de manera sencilla con la aplicación. Durante esta etapa se solucionaron los problemas de portabilidad a dispositivos móviles basados en Android y se optimizó el rendimiento incorporando límites en la tasa de fotogramas procesados y métodos de escalado digital.

Finalmente, Se realizaron pruebas de evaluación en entornos reales. Estas pruebas sirvieron para evaluar la precisión del sistema validando su eficacia a la hora de procesar imágenes marítimas de forma dinámica, lo que confirma el buen funcionamiento global de la aplicación.

A nivel metodológico, el desarrollo de este proyecto ha representado un desafío, debido principalmente a la exigente curva de aprendizaje asociada al uso de plataformas como Unity y Jupyter, así como a la necesidad de trabajar de forma simultánea con distintos lenguajes de programación, entre ellos C# y Python. Con el objetivo de superar estas dificultades técnicas y optimizar el tiempo disponible, se recurrió de forma habitual al apoyo de asistentes de inteligencia artificial en la estructuración y desarrollo del código, adoptando un enfoque de trabajo conocido actualmente como *vibe coding*.

Esta forma de desarrollo ha supuesto un cambio significativo en la manera de abordar el proyecto. En lugar de centrar el esfuerzo en la memorización de sintaxis o en la corrección manual de errores, el trabajo se ha orientado hacia aspectos de mayor nivel, como el diseño de la arquitectura del sistema, la supervisión de la integración de los modelos de visión artificial y la resolución de problemas complejos.

Finalmente, a título personal, el desarrollo de este TFG me ha servido para sumergirme más a fondo en el ámbito de la visión artificial. El trabajo me ha aportado una visión del proceso de desarrollo de una

aplicación desde cero, que me ha enseñado el rigor y el esfuerzo técnico necesario para conseguir un resultado óptimo y satisfactorio.

5.2 Líneas futuras

Habiendo desarrollado la aplicación de detección y clasificación de embarcaciones en Android mediante arquitectura YOLO y Unity se exponen una serie de líneas futuras que permiten continuar con el desarrollo de este trabajo y aplicar mejoras sobre el mismo siguiendo la línea de trabajos mencionada en la introducción:

1. Mejora de la precisión del modelo de detección mediante el uso de *datasets* más amplios y variados, incorporando condiciones adversas como: niebla, lluvia o baja iluminación,
2. Integración completa con sistemas AIS, permitiendo la fusión de datos entre detección visual y datos de posicionamiento, con el fin de mejorar la identificación y seguimiento de embarcaciones.
3. Implementación de mejoras en el sistema de tracking que permitan mantener la identidad de los objetos detectados de forma más eficaz en escenarios con múltiples embarcaciones y oclusiones.
4. Desarrollo de un sistema de clasificación más detallado que permita diferenciar un mayor número de tipos de embarcaciones, incluyendo categorías específicas de interés operativo.
5. Mejora del interfaz de usuario, adaptando la visualización de los marcadores y la información mostrada para facilitar la interpretación.
6. Implementación del sistema en otras plataformas, como iOS o dispositivos de realidad aumentada, ampliando su portabilidad.
7. Integración del sistema en aplicaciones reales de vigilancia marítima, incluyendo su uso en drones, sistemas costeros o plataformas embarcadas.

6 BIBLIOGRAFÍA

- [1] Armada, «Seguridad Marítima - La Armada, un actor indispensable - Especiales - Armada - Ministerio de Defensa - Gobierno de España». Accedido: 22 de marzo de 2026. [En línea]. Disponible en: <https://armada.defensa.gob.es/ArmadaPortal/page/Portal/ArmadaEspañola/conocenos/especiales/refLang-es/05actividades--seguridadmaritima>
- [2] «Inicio - Documentos de Ultralytics YOLO». Accedido: 22 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/es/>
- [3] S. Liang *et al.*, «Object Detectors in the Open Environment: Challenges, Solutions, and Outlook», 9 de abril de 2024, *arXiv*: arXiv:2403.16271. doi: 10.48550/arXiv.2403.16271.
- [4] «Desarrollo de aplicación para la identificación de embarcaciones y elementos del entorno marítimo mediante realidad aumentada». Accedido: 22 de marzo de 2026. [En línea]. Disponible en: <https://calderon.cud.uvigo.es/items/27ebb1ab-792f-43fb-b848-9a065e9c4ed4>
- [5] P. Díaz García y M. (advisor) Rodelgo Lacruz, «Desarrollo de una aplicación de Realidad Aumentada para la representación de datos AIS en tiempo real», 2025, Accedido: 22 de marzo de 2026. [En línea]. Disponible en: <https://calderon.cud.uvigo.es/handle/123456789/924>
- [6] R. Lacruz, D. García, N. Nieto, T. Pastoriza, y P. Luna, «Visualización de Datos AIS en Tiempo Real mediante Realidad Aumentada».
- [7] B. Gonçalves, «The Turing Test is a Thought Experiment», *Minds & Machines*, vol. 33, n.º 1, pp. 1-31, mar. 2023, doi: 10.1007/s11023-022-09616-8.
- [8] M. Giletta, A. Giordano, y N. Mercaú, «Inteligencia Artificial: definiciones en disputa».
- [9] I. Gupta y G. Nagpal, *Artificial Intelligence and Expert Systems*. Walter de Gruyter GmbH & Co KG, 2020.
- [10] W. S. McCulloch y W. Pitts, «A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY».
- [11] W. Zamora-Cárdenas, M. Zumbado, y I. Trejos-Zelaya, «McCulloch-Pitts Artificial Neuron and Rosenblatt's Perceptron: An abstract specification in Z», *Technology Inside by CPIC*, vol. 5, pp. 16-29, jul. 2020.
- [12] «Deep learning for AI», *Communications of the ACM*, Accedido: 22 de marzo de 2026. [En línea]. Disponible en: <https://dl.acm.org/doi/10.1145/3448250>
- [13] K. M. Y. Ali, S. Akter, S. Islam, y M. E. Mridha, «Advancing Computational Intelligence: AI-Based Algorithm Design and Optimization in Programming», *Journal of Computer Science and Technology Studies*, vol. 7, n.º 1, pp. 122-138, feb. 2025, doi: 10.32996/jcsts.2025.7.1.10.
- [14] A. Thakur y A. Konde, «Fundamentals of neural networks», *International Journal for Research in Applied Science and Engineering Technology*, vol. 9, n.º VIII, pp. 407-426, 2021.

- [15] R. Qamar y B. A. Zardari, «Artificial Neural Networks: An Overview», *Mesopotamian Journal of Computer Science*, vol. 2023, pp. 124-133, ago. 2023, doi: 10.58496/MJCSC/2023/015.
- [16] Y. Matsuzaka y R. Yashiro, «AI-Based Computer Vision Techniques and Expert Systems», *AI*, vol. 4, n.º 1, pp. 289-302, mar. 2023, doi: 10.3390/ai4010013.
- [17] D. Onugha, «Image Augmentation: A Fun and Easy Way to Improve Computer Vision Models», Comet. Accedido: 22 de marzo de 2026. [En línea]. Disponible en: <https://www.comet.com/site/blog/image-augmentation-a-fun-and-easy-way-to-improve-computer-vision-models>
- [18] M. Nixon y A. S. Aguado, *Feature Extraction and Image Processing for Computer Vision*. Academic Press, 2025.
- [19] «Why FPS Matters: Computer Vision Frame Rate Guide». Accedido: 22 de marzo de 2026. [En línea]. Disponible en: <https://www.ultralytics.com/blog/understanding-the-role-of-fps-in-computer-vision>
- [20] S. V. Mahadevkar *et al.*, «A Review on Machine Learning Styles in Computer Vision—Techniques and Future Directions», *IEEE Access*, vol. 10, pp. 107293-107329, 2022, doi: 10.1109/ACCESS.2022.3209825.
- [21] S. Ghosh, N. Das, I. Das, y U. Maulik, «Understanding Deep Learning Techniques for Image Segmentation», *ACM Comput. Surv.*, vol. 52, n.º 4, p. 73:1-73:35, ago. 2019, doi: 10.1145/3329784.
- [22] R. Szeliski, *Computer Vision: Algorithms and Applications*. Springer Nature, 2022.
- [23] J. Chai, H. Zeng, A. Li, y E. W. T. Ngai, «Deep learning in computer vision: A critical review of emerging techniques and application scenarios», *Machine Learning with Applications*, vol. 6, p. 100134, dic. 2021, doi: 10.1016/j.mlwa.2021.100134.
- [24] «Explicación de R-CNN: Detección de objetos | Ultralytics». Accedido: 22 de marzo de 2026. [En línea]. Disponible en: <https://www.ultralytics.com/es/blog/what-is-r-cnn-a-quick-overview>
- [25] T.-Y. Lin, P. Goyal, R. Girshick, K. He, y P. Dollár, «Focal Loss for Dense Object Detection», 7 de febrero de 2018, *arXiv*: arXiv:1708.02002. doi: 10.48550/arXiv.1708.02002.
- [26] J. Hui, «SSD object detection: Single Shot MultiBox Detector for real-time processing», Medium. Accedido: 22 de marzo de 2026. [En línea]. Disponible en: <https://jonathan-hui.medium.com/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06>
- [27] «Home - Ultralytics YOLO Docs». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/>
- [28] M. Mao y M. Hong, «YOLO Object Detection for Real-Time Fabric Defect Inspection in the Textile Industry: A Review of YOLOv1 to YOLOv11», *Sensors*, vol. 25, n.º 7, p. 2270, ene. 2025, doi: 10.3390/s25072270.
- [29] N. Mittal, A. Vaidya, y A. S. Kapoor, «Object Detection and Classification Using Yolo», *International Journal of Scientific Research*, vol. 5, n.º 2, 2019.
- [30] S. Jamgaonkar, J. S. Gowda, S. S. Chouhan, R. K. Patel, y A. Pandey, «An analysis of different YOLO models for real-time object detection», en *2024 4th international conference on sustainable expert systems (ICSES)*, IEEE, 2024, pp. 951-955. Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://ieeexplore.ieee.org/abstract/document/10763020/>
- [31] Ultralytics, «Models Supported by Ultralytics». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/models/>
- [32] Ultralytics, «YOLOv3, and YOLOv3u». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/models/yolov3/>
- [33] Ultralytics, «YOLOv4: High-Speed and Precise Object Detection». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/models/yolov4/>
- [34] Ultralytics, «Ultralytics YOLOv5». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/models/yolov5/>
- [35] Ultralytics, «YOLOv7: Trainable Bag-of-Freebies». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/models/yolov7/>

- [36] Ultralytics, «Meituan YOLOv6». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/models/yolov6/>
- [37] Ultralytics, «Explore Ultralytics YOLOv8». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/models/yolov8/>
- [38] Ultralytics, «YOLOv9: A Leap Forward in Object Detection Technology». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/models/yolov9/>
- [39] Ultralytics, «YOLOv10: Real-Time End-to-End Object Detection». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/models/yolov10/>
- [40] Ultralytics, «Ultralytics YOLO11». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/models/yolo11/>
- [41] Ultralytics, «Ultralytics YOLO26». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/models/yolo26/>
- [42] «Object Detection - Ultralytics YOLO Docs». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://docs.ultralytics.com/tasks/detect/#models>
- [43] «Best Computer Vision Datasets 2025 | Ultralytics». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://www.ultralytics.com/blog/exploring-the-best-computer-vision-datasets-in-2025>
- [44] «COCO - Common Objects in Context». Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://cocodataset.org/#home>
- [45] xdobro23, «YOLOv8-16bit/coco.md at c9be1f3cce89778f79fb462797b8ca0300e3813d», Gitea: Git with a cup of tea. Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://strade.fit.vutbr.cz/git/git/xdobro23/YOLOv8-16bit/src/commit/c9be1f3cce89778f79fb462797b8ca0300e3813d/docs/datasets/pose/coco.md>
- [46] T.-Y. Lin *et al.*, «Microsoft COCO: Common Objects in Context», 21 de febrero de 2015, *arXiv*: arXiv:1405.0312. doi: 10.48550/arXiv.1405.0312.
- [47] E. Gundogdu, B. Solmaz, V. Yücesoy, y A. Koç, «MARVEL: A Large-Scale Image Dataset for Maritime Vessels», en *Computer Vision – ACCV 2016*, S.-H. Lai, V. Lepetit, K. Nishino, y Y. Sato, Eds., Cham: Springer International Publishing, 2017, pp. 165-180. doi: 10.1007/978-3-319-54193-8_11.
- [48] Jiaming-Wang, *jiaming-wang/SeaShips*. (13 de marzo de 2026). Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://github.com/jiaming-wang/SeaShips>
- [49] zzndream, *zzndream/ShipRSImageNet*. (21 de marzo de 2026). Python. Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://github.com/zzndream/ShipRSImageNet>
- [50] J. Manesis, *jasonmanesis/Satellite-Imagery-Datasets-Containing-Ships*. (19 de marzo de 2026). Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://github.com/jasonmanesis/Satellite-Imagery-Datasets-Containing-Ships>
- [51] «Unreal Engine | Funciones», Unreal Engine. Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://www.unrealengine.com/es-ES/features>
- [52] «Introducción a Android Studio», Android Developers. Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://developer.android.com/studio/intro?hl=es-419>
- [53] «Recorrido del producto de Unity Industry: Demos y tutoriales», Unity. Accedido: 23 de marzo de 2026. [En línea]. Disponible en: <https://unity.com/campaign/unity-industry-product-walkthrough-demos>
- [54] «Kyoto by Another Kid & Pratzapp | Free To Use». Accedido: 26 de marzo de 2026. [En línea]. Disponible en: <https://freetouse.com/music/another-kid/kyoto>
- [55] «Seaships7000 Object Detection Dataset by ships», Roboflow. Accedido: 25 de marzo de 2026. [En línea]. Disponible en: <https://universe.roboflow.com/ships-tznqe/seaships7000-yxbuv>

ANEXO I: SCRIPT YOLODETECTOR.CS

```

1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.UI;
5 using UnityEngine.Android;
6 using System.Linq;
7 using UnityEngine.SceneManagement;
8
9 public class YoloDetector : MonoBehaviour
10 {
11     [Header("Configuración")]
12     public Unity.InferenceEngine.ModelAsset modelAsset;
13     public RawImage displayScreen;
14     public GameObject boxPrefab;
15     public Transform boxContainer;
16
17     [Header("Ajustes de Detección")]
18     [Range(0f, 1f)] public float confidenceThreshold = 0.5f;
19     [Range(0f, 1f)] public float iouThreshold = 0.4f;
20
21     [Header("Rendimiento")]
22     [Range(1, 30)] public int detectEveryNFrames = 15;
23
24     [Header("Tracking (Seguimiento)")]
25     public float maxTrackingDistance = 150f;
26     public int maxLostFrames = 3;
27
28     [Header("Control de Zoom")]
29     public Slider zoomSlider;
30     private float iaZoomLevel = 1f;
31
32     [Header("UI")]
33     public Image detectionPanel;
34     public TMPro.TextMeshProUGUI detectionLabel;
35     public Image fadeOverlay;
36     public float fadeInDuration = 0.5f;
37
38     [Header("Menú de Opciones")]
39     public string optionsSceneName = "OptionsScene";
40
41     [Header("Config externa")]
42     public DetectorConfig config;
43
44     private Color colorDetected = new Color(0.1f, 0.8f, 0.1f, 1f);
45     private Color colorNoDetected = new Color(0.8f, 0.1f, 0.1f, 1f);
46
47     private Unity.InferenceEngine.Model runtimeModel;
48     private Unity.InferenceEngine.Worker worker;

```

```
49     private WebCamTexture webcam;
50     private RenderTexture renderTexture;
51
52     private Unity.InferenceEngine.Tensor<float> inputTensor;
53     private Unity.InferenceEngine.TextureTransform inputTransform;
54
55     private List<GameObject> boxPool = new List<GameObject>();
56     private int frameCounter = 0;
57
58     private bool inferencePending = false;
59     private Unity.InferenceEngine.Tensor<float> outputTensor;
60
61     private bool isReady = false;
62     private ScreenOrientation lastOrientation =
63 ScreenOrientation.AutoRotation;
64
65     const int IMAGE_SIZE = 640;
66     const int NUM_CLASSES = 6;
67
68     public readonly string[] SHIP_CLASSES = {
69         "Bulk Cargo Carrier",
70         "Container Ship",
71         "Fishing Boat",
72         "General Cargo Ship",
73         "Ore Carrier",
74         "Passenger Ship"
75     };
76
77     public struct Detection { public float x, y, w, h, score; public int
78 classIndex; }
79
80     public class TrackedObject
81     {
82         public int id;
83         public Detection box;
84         public int lostFrames;
85     }
86
87     private List<TrackedObject> trackedObjects = new List<TrackedObject>();
88     private int nextTrackId = 1;
89
90     // — INICIO
91
92     IEnumerator Start()
93     {
94         if (fadeOverlay != null)
95         {
96             fadeOverlay.color = new Color(0, 0, 0, 1f);
97             fadeOverlay.raycastTarget = false;
98         }
```

```

99
100     Screen.orientation = ScreenOrientation.LandscapeLeft;
101     Screen.autorotateToPortrait = false;
102     Screen.autorotateToPortraitUpsideDown = false;
103     Screen.autorotateToLandscapeLeft = true;
104     Screen.autorotateToLandscapeRight = true;
105
106     lastOrientation = Screen.orientation;
107
108     if (zoomSlider != null)
109     {
110         zoomSlider.minValue = 1f;
111         zoomSlider.maxValue = 5f;
112         zoomSlider.value = 1f;
113         zoomSlider.onValueChanged.AddListener(SetZoomLevel);
114     }
115
116     yield return new WaitForEndOfFrame();
117     yield return new WaitForEndOfFrame();
118
119 #if UNITY_ANDROID
120     if (!Permission.HasUserAuthorizedPermission(Permission.Camera))
121     {
122         bool permissionResolved = false;
123         var callbacks = new PermissionCallbacks();
124         callbacks.PermissionGranted += (perm) => { permissionResolved =
125 true; };
126         callbacks.PermissionDenied += (perm) => { permissionResolved =
127 true; };
128         Permission.RequestUserPermission(Permission.Camera, callbacks);
129         yield return new WaitUntil(() => permissionResolved);
130     }
131 #endif
132
133     runtimeModel = Unity.InferenceEngine.ModelLoader.Load(modelAsset);
134     worker = new Unity.InferenceEngine.Worker(runtimeModel,
135 Unity.InferenceEngine.BackendType.CPU);
136
137     WebCamDevice[] devices = WebCamTexture.devices;
138     if (devices.Length > 0)
139     {
140         string camName = devices.FirstOrDefault(d =>
141 !d.isFrontFacing).name ?? devices[0].name;
142         webcam = new WebCamTexture(camName, 1280, 720, 30);
143         webcam.Play();
144
145         yield return new WaitUntil(() => webcam.width > 16 &&
146 webcam.didUpdateThisFrame);
147         yield return new WaitForEndOfFrame();
148

```

```
149         if (displayScreen != null)
150         {
151             displayScreen.texture = webcam;
152             AdjustDisplay();
153         }
154     }
155
156     UpdateDetectionUI(0);
157
158     renderTexture = new RenderTexture(IMAGE_SIZE, IMAGE_SIZE, 0,
159 RenderTextureFormat.ARGB32);
160     renderTexture.Create();
161
162     inputTransform = new
163 Unity.InferenceEngine.TextureTransform().SetDimensions(IMAGE_SIZE,
164 IMAGE_SIZE, 3);
165     inputTensor = new Unity.InferenceEngine.Tensor<float>(new
166 Unity.InferenceEngine.TensorShape(1, 3, IMAGE_SIZE, IMAGE_SIZE));
167
168     isReady = true;
169
170     if (DetectorConfig.Instance != null)
171     {
172         config = DetectorConfig.Instance;
173         confidenceThreshold = config.confidenceThreshold;
174         iouThreshold = config.iouThreshold;
175         detectEveryNFrames = config.detectEveryNFrames;
176         maxTrackingDistance = config.maxTrackingDistance;
177         maxLostFrames = config.maxLostFrames;
178     }
179
180     if (fadeOverlay != null)
181         StartCoroutine(FadeInFromBlack());
182 }
183
184 IEnumerator FadeInFromBlack()
185 {
186     float elapsed = 0f;
187     while (elapsed < fadeInDuration)
188     {
189         elapsed += Time.deltaTime;
190         float alpha = Mathf.Lerp(1f, 0f, elapsed / fadeInDuration);
191         fadeOverlay.color = new Color(0, 0, 0, alpha);
192         yield return null;
193     }
194     fadeOverlay.color = new Color(0, 0, 0, 0f);
195 }
196
197 void AdjustDisplay()
198 {
```

```

199     if (displayScreen == null || webcam == null) return;
200
201     float screenW = Screen.width;
202     float screenH = Screen.height;
203
204     if (screenW < screenH)
205     {
206         float tmp = screenW;
207         screenW = screenH;
208         screenH = tmp;
209     }
210
211     RectTransform rt = displayScreen.rectTransform;
212     rt.localEulerAngles = Vector3.zero;
213     rt.localScale = Vector3.one;
214
215     rt.anchorMin = new Vector2(0.5f, 0.5f);
216     rt.anchorMax = new Vector2(0.5f, 0.5f);
217     rt.pivot = new Vector2(0.5f, 0.5f);
218     rt.anchoredPosition = Vector2.zero;
219
220     Canvas canvas = displayScreen.canvas;
221     float scaleFactor = canvas != null ? canvas.scaleFactor : 1f;
222     float uiW = screenW / scaleFactor;
223     float uiH = screenH / scaleFactor;
224
225     float camAspect = (float)webcam.width / (float)webcam.height;
226
227     float targetW, targetH;
228     if (uiW / uiH > camAspect)
229     {
230         targetW = uiW;
231         targetH = uiW / camAspect;
232     }
233     else
234     {
235         targetH = uiH;
236         targetW = uiH * camAspect;
237     }
238
239     rt.sizeDelta = new Vector2(targetW, targetH);
240
241     var fitter = displayScreen.GetComponent<AspectRatioFitter>();
242     if (fitter != null) Destroy(fitter);
243
244     AplicarZoomVisual();
245 }
246
247 void Update()
248 {

```

```
249         if (!isReady || webcam == null || !webcam.isPlaying) return;
250
251         if (Screen.orientation != lastOrientation)
252         {
253             lastOrientation = Screen.orientation;
254             AdjustDisplay();
255         }
256
257         if (inferencePending && outputTensor != null)
258         {
259             if (outputTensor.IsReadbackRequestDone())
260             {
261                 using var cpuOutput = outputTensor.ReadbackAndClone();
262                 ParseYoloOutput(cpuOutput);
263                 inferencePending = false;
264             }
265             return;
266         }
267
268         frameCounter++;
269         if (frameCounter >= detectEveryNFrames)
270         {
271             frameCounter = 0;
272
273             float scale = 1f / iaZoomLevel;
274             float offset = (1f - scale) / 2f;
275
276             Vector2 cropScale = new Vector2(scale, scale);
277             Vector2 cropOffset = new Vector2(offset, offset);
278
279             Graphics.Blit(webcam, renderTexture, cropScale, cropOffset);
280
281             Unity.InferenceEngine.TextureConverter.ToTensor(renderTexture,
282 inputTensor, inputTransform);
283             worker.Schedule(inputTensor);
284
285             outputTensor = worker.PeekOutput() as
286 Unity.InferenceEngine.Tensor<float>;
287             if (outputTensor != null)
288             {
289                 outputTensor.ReadbackRequest();
290                 inferencePending = true;
291             }
292         }
293     }
294
295     public void SetZoomLevel(float nuevoZoom)
296     {
297         iaZoomLevel = Mathf.Clamp(nuevoZoom, 1f, 5f);
298         AplicarZoomVisual();
```

```

299     }
300
301     void AplicarZoomVisual()
302     {
303         if (displayScreen != null && webcam != null)
304         {
305             bool isMirrored = webcam.videoVerticallyMirrored;
306
307             float uvScale = 1f / iaZoomLevel;
308             float uvOffset = (1f - uvScale) / 2f;
309
310             Rect newUvRect;
311
312             if (isMirrored)
313             {
314                 newUvRect = new Rect(1f - uvOffset, uvOffset, -uvScale,
315 uvScale);
316             }
317             else
318             {
319                 newUvRect = new Rect(uvOffset, uvOffset, uvScale, uvScale);
320             }
321
322             displayScreen.uvRect = newUvRect;
323         }
324     }
325
326     void ParseYoloOutput(Unity.InferenceEngine.Tensor<float> output)
327     {
328         List<Detection> rawDetections = new List<Detection>();
329         int numBoxes = output.shape[2];
330
331         for (int i = 0; i < numBoxes; i++)
332         {
333             float maxScore = 0f;
334             int bestClassIndex = -1;
335
336             for (int c = 0; c < NUM_CLASSES; c++)
337             {
338                 float score = output[0, 4 + c, i];
339                 if (score > maxScore)
340                 {
341                     maxScore = score;
342                     bestClassIndex = c;
343                 }
344             }
345
346             if (maxScore > confidenceThreshold)
347             {
348                 rawDetections.Add(new Detection

```

```
349         {
350             x = output[0, 0, i],
351             y = output[0, 1, i],
352             w = output[0, 2, i],
353             h = output[0, 3, i],
354             score = maxScore,
355             classIndex = bestClassIndex
356         });
357     }
358 }
359
360 var nmsDetections = NMS(rawDetections);
361 UpdateTracking(nmsDetections);
362
363 int activeBoats = trackedObjects.Count(t => t.lostFrames == 0);
364 UpdateDetectionUI(activeBoats);
365
366 DrawBoxes(trackedObjects);
367 }
368
369 void UpdateTracking(List<Detection> currentDetections)
370 {
371     List<TrackedObject> nextTrackedList = new List<TrackedObject>();
372     List<Detection> unassignedDetections = new
373 List<Detection>(currentDetections);
374
375     foreach (var track in trackedObjects)
376     {
377         float bestDist = maxTrackingDistance;
378         int bestIdx = -1;
379
380         for (int i = 0; i < unassignedDetections.Count; i++)
381         {
382             var det = unassignedDetections[i];
383             float dist = Vector2.Distance(
384                 new Vector2(track.box.x, track.box.y),
385                 new Vector2(det.x, det.y)
386             );
387
388             if (dist < bestDist)
389             {
390                 bestDist = dist;
391                 bestIdx = i;
392             }
393         }
394
395         if (bestIdx != -1)
396         {
397             track.box = unassignedDetections[bestIdx];
398             track.lostFrames = 0;
```

```

399         nextTrackedList.Add(track);
400         unassignedDetections.RemoveAt(bestIdx);
401     }
402     else
403     {
404         track.lostFrames++;
405         if (track.lostFrames <= maxLostFrames)
406         {
407             nextTrackedList.Add(track);
408         }
409     }
410 }
411
412 foreach (var newDet in unassignedDetections)
413 {
414     nextTrackedList.Add(new TrackedObject
415     {
416         id = nextTrackId++,
417         box = newDet,
418         lostFrames = 0
419     });
420 }
421
422 trackedObjects = nextTrackedList;
423 }
424
425 void UpdateDetectionUI(int count)
426 {
427     if (detectionPanel == null || detectionLabel == null) return;
428     if (count > 0)
429     {
430         detectionPanel.color = colorDetected;
431         detectionLabel.text = count == 1 ? "1 BARCO DETECTADO" :
432 $"{count} BARCOS DETECTADOS";
433     }
434     else
435     {
436         detectionPanel.color = colorNoDetected;
437         detectionLabel.text = "SIN DETECCIÓN";
438     }
439 }
440
441 void DrawBoxes(List<TrackedObject> trackedList)
442 {
443     HideAllBoxes();
444
445     if (displayScreen == null || boxContainer == null || boxPrefab ==
446 null) return;
447
448     float scaleX = displayScreen.rectTransform.rect.width / IMAGE_SIZE;

```

```
449         float scaleY = displayScreen.rectTransform.rect.height / IMAGE_SIZE;
450
451         foreach (var track in trackedList)
452         {
453             if (track.lostFrames > 0) continue;
454
455             GameObject box = GetOrCreateBox();
456             RectTransform rt = box.GetComponent<RectTransform>();
457             if (rt == null) continue;
458
459             rt.anchoredPosition = new Vector2(
460                 (track.box.x - IMAGE_SIZE / 2f) * scaleX,
461                 -(track.box.y - IMAGE_SIZE / 2f) * scaleY
462             );
463             rt.sizeDelta = new Vector2(track.box.w * scaleX, track.box.h *
464 scaleY);
465
466         }
467     }
468
469     void HideAllBoxes()
470     {
471         foreach (var box in boxPool) box.SetActive(false);
472     }
473
474     GameObject GetOrCreateBox()
475     {
476         foreach (var box in boxPool)
477             if (!box.activeSelf)
478             {
479                 box.SetActive(true);
480                 return box;
481             }
482
483         var newBox = Instantiate(boxPrefab, boxContainer);
484         boxPool.Add(newBox);
485         return newBox;
486     }
487
488     List<Detection> NMS(List<Detection> inputList)
489     {
490         var results = new List<Detection>();
491         var sortedList = inputList.OrderByDescending(d => d.score).ToList();
492         while (sortedList.Count > 0)
493         {
494             var best = sortedList[0];
495             results.Add(best);
496             sortedList.RemoveAt(0);
497             for (int i = sortedList.Count - 1; i >= 0; i--)
498                 if (CalculateIoU(best, sortedList[i]) > iouThreshold)
```

```
499             sortedList.RemoveAt(i);
500         }
501         return results;
502     }
503
504     float CalculateIoU(Detection a, Detection b)
505     {
506         float xA = Mathf.Max(a.x - a.w / 2, b.x - b.w / 2);
507         float yA = Mathf.Max(a.y - a.h / 2, b.y - b.h / 2);
508         float xB = Mathf.Min(a.x + a.w / 2, b.x + b.w / 2);
509         float yB = Mathf.Min(a.y + a.h / 2, b.y + b.h / 2);
510         float interArea = Mathf.Max(0, xB - xA) * Mathf.Max(0, yB - yA);
511         return interArea / (a.w * a.h + b.w * b.h - interArea);
512     }
513
514     private void OnDestroy()
515     {
516         isReady = false;
517         inferencePending = false;
518         webcam?.Stop();
519         renderTexture?.Release();
520         inputTensor?.Dispose();
521         worker?.Dispose();
522         boxPool.Clear();
523     }
524
525     public void OpenOptionsMenu()
526     {
527         SceneManager.LoadScene(optionsSceneName);
528     }
529 }
```


ANEXO II: SCRIPT OPTIONMENUCONTROLLER.CS

```

1  using UnityEngine;
2  using UnityEngine.UI;
3  using TMPro;
4  using UnityEngine.SceneManagement;
5
6  public class DetectorOptionsMenu : MonoBehaviour
7  {
8      [Header("UI Sliders")]
9      public Slider confidenceSlider;
10     public TextMeshProUGUI confidenceLabel;
11
12     public Slider iouSlider;
13     public TextMeshProUGUI iouLabel;
14
15     public Slider framesSlider;
16     public TextMeshProUGUI framesLabel;
17
18     [Header("Botón Volver")]
19     public Button volverButton;
20
21     [Header("Escenas")]
22     public string detectorSceneName = "DetectorScene";
23
24     DetectorConfig config;
25
26     void Start()
27     {
28         config = DetectorConfig.Instance;
29         if (config == null)
30         {
31             Debug.LogError(";No hay DetectorConfig! Crea el objeto en
32 StartScreen.");
33             return;
34         }
35
36         // Configurar sliders
37         confidenceSlider.minValue = 0f; confidenceSlider.maxValue = 1f;
38         iouSlider.minValue = 0f; iouSlider.maxValue = 1f;
39         framesSlider.minValue = 1f; framesSlider.maxValue = 30f;
40         framesSlider.wholeNumbers = true;
41
42         // Inicializar con valores actuales
43         confidenceSlider.value = config.confidenceThreshold;
44         iouSlider.value = config.iouThreshold;
45         framesSlider.value = config.detectEveryNFrames;
46

```

```
47         UpdateLabels();
48
49         // Suscribirse a cambios
50         confidenceSlider.onValueChanged.AddListener(v => {
51             config.confidenceThreshold = v;
52             UpdateLabels();
53         });
54
55         iouSlider.onValueChanged.AddListener(v => {
56             config.iouThreshold = v;
57             UpdateLabels();
58         });
59
60         framesSlider.onValueChanged.AddListener(v => {
61             config.detectEveryNFrames = (int)v;
62             UpdateLabels();
63         });
64
65         // Botón volver
66         if (volverButton != null)
67             volverButton.onClick.AddListener(VolverAlDetector);
68     }
69
70     void UpdateLabels()
71     {
72         confidenceLabel.text = $"Confianza: {config.confidenceThreshold:F2}";
73         iouLabel.text = $"IoU: {config.iouThreshold:F2}";
74         framesLabel.text = $"Cada {config.detectEveryNFrames} frames";
75     }
76
77     public void VolverAlDetector()
78     {
79         SceneManager.LoadScene(detectorSceneName);
80     }
81
82
83     public void SalirDeLaApp()
84     {
85         Debug.Log("Saliendo de la aplicación..."); // Solo visible en el Editor
86         Application.Quit(); // Cierra el juego al estar compilado en Android/PC
87     }
88 }
```

ANEXO III: SCRIPT STARTSCREENCONTROLLER.CS

```

1 using UnityEngine;
2 using UnityEngine.UI;
3 using UnityEngine.SceneManagement;
4 using System.Collections;
5
6 public class StartScreenController : MonoBehaviour
7 {
8     [Header("UI Principal")]
9     public Button startButton;
10    public Image fadeOverlay;
11
12    [Header("UI Audio")]
13    public Button audioButton;           // El botón clickeable
14    public Image audioButtonImage;       // La imagen que cambiará
15    public Sprite iconAudioOn;           // Imagen de la bocina encendida
16    public Sprite iconAudioOff;          // Imagen de la bocina apagada
17
18    [Header("Música")]
19    public AudioSource backgroundMusic; // El reproductor de música
20
21    [Header("Ajustes")]
22    public string detectorSceneName = "DetectorScene";
23    public float fadeOutDuration = 0.5f;
24
25    private bool isLoading = false;
26    private bool isAudioOn = true;
27
28    void Start()
29    {
30        Screen.orientation = ScreenOrientation.LandscapeLeft;
31
32        if (fadeOverlay != null)
33        {
34            fadeOverlay.color = new Color(0, 0, 0, 0);
35            fadeOverlay.raycastTarget = false;
36        }
37
38        // Asignar los eventos a los botones
39        if (startButton != null)
40            startButton.onClick.AddListener(OnStartPressed);
41
42        if (audioButton != null)
43            audioButton.onClick.AddListener(ToggleAudio);
44
45        // Poner el icono correcto al inicio
46        UpdateAudioIcon();
47    }
48

```

```
49     void ToggleAudio()
50     {
51         isAudioOn = !isAudioOn; // Invierte el estado (de true a false o
52 viceversa)
53
54         if (backgroundMusic != null)
55             backgroundMusic.mute = !isAudioOn; // Mutea o desmutea
56
57         UpdateAudioIcon(); // Cambia la imagen
58     }
59
60     void UpdateAudioIcon()
61     {
62         if (audioButtonImage != null)
63         {
64             audioButtonImage.sprite = isAudioOn ? iconAudioOn : iconAudioOff;
65         }
66     }
67
68     void OnStartPressed()
69     {
70         if (isLoading) return;
71         isLoading = true;
72
73         // Ocultar botones inmediatamente al hacer clic
74         if (startButton != null) startButton.gameObject.SetActive(false);
75         if (audioButton != null) audioButton.gameObject.SetActive(false);
76
77         if (fadeOverlay != null) fadeOverlay.raycastTarget = true;
78
79         StartCoroutine(FadeToBlackAndLoad());
80     }
81
82     IEnumerator FadeToBlackAndLoad()
83     {
84         float elapsed = 0f;
85
86         // Guardamos el volumen inicial para ir bajándolo
87         float startVolume = isAudioOn && backgroundMusic != null ?
88 backgroundMusic.volume : 0f;
89
90         // Fundido a negro (pantalla) y bajada de volumen (música)
91         while (elapsed < fadeOutDuration)
92         {
93             elapsed += Time.deltaTime;
94             float t = elapsed / fadeOutDuration;
95
96             // Fade visual
97             fadeOverlay.color = new Color(0, 0, 0, Mathf.Lerp(0f, 1f, t));
98         }
```

```
99         // Fade audio
100         if (backgroundMusic != null && isAudioOn)
101         {
102             backgroundMusic.volume = Mathf.Lerp(startVolume, 0f, t);
103         }
104
105         yield return null;
106     }
107
108     fadeOverlay.color = new Color(0, 0, 0, 1f);
109     if (backgroundMusic != null) backgroundMusic.volume = 0f;
110
111     yield return new WaitForSeconds(0.2f);
112     SceneManager.LoadSceneAsync(detectorSceneName);
113 }
114 }
115
```


ANEXO IV: SCRIPT SERVIDOR GPU

```

1 import os
2 from pathlib import Path
3
4 # 1. Entra en tu carpeta del dataset
5 dataset_dir = Path('SeaShips.v1-seaships.yolov11')
6 os.chdir(dataset_dir)
7 print(f"📁 Directorio actual: {os.getcwd()}")
8
9 # 2. Verifica estructura (debe mostrar train/valid/test/)
10 !ls -la
11 !ls train/images/ | wc -l      # Cuenta imágenes train
12 !head -5 data.yaml            # Muestra YAML
13
14 # 3. Fix automático del YAML (cambia ../ por ./)
15 with open('data.yaml', 'r') as f:
16     content = f.read()
17 content = content.replace('../train/images',
18 'train/images').replace('../valid/images',
19 'valid/images').replace('../test/images', 'test/images')
20 with open('data_fixed.yaml', 'w') as f:
21     f.write(content)
22 print("✅ YAML corregido: data_fixed.yaml")
23
24 # 4. ¡ENTRENA YOLO11N!
25 from ultralytics import YOLO
26 model = YOLO('yololln.pt')      # Nano preentrenado
27 results = model.train(
28     data='data_fixed.yaml',
29     epochs=100,
30     imgsz=640,
31     device=0,      # RTX 3090 Ti
32     batch=32,      # Perfecto para 24GB VRAM
33     name='seaships_yololln',
34     patience=20    # Early stopping
35 )
36 from ultralytics import YOLO
37
38 # Carga el modelo
39 model = YOLO('runs/detect/seaships_yololln/weights/best.pt')
40
41 # EXPORTA ONNX
42 success = model.export(
43     format='onnx',
44     imgsz=640,      # Tu tamaño entrenamiento
45     dynamic=False,  # Fijo = 2x más rápido
46     simplify=True,  # Optimiza grafo
47     opset=12,       # Compatible everywhere
48     workspace=4     # 4GB VRAM ok para 3090 Ti
49 )
50
51 print(f"✅ ONNX creado: {model.model[-1].export_dir}/best.onnx")
52 from ultralytics import YOLO
53 import subprocess

```

```
54
55 model = YOLO('runs/detect/seaships_yolo11n/weights/best.pt')
56
57 # 1. Encuentra primera foto test (CORRECTO)
58 test_files = subprocess.check_output("ls test/images/ | head -1",
59 shell=True).decode().strip()
60 test_img = f'test/images/{test_files}'
61 print(f"🖼️ Testeando: {test_img}")
62
63 # 2. DETECTA barcos
64 results = model(
65     source=test_img,
66     conf=0.25,
67     save=True,
68     show=True,
69     project='results_demo',
70     name='test_ship'
71 )
72
73 # 3. Muestra resultados
74 print("\n🚢 BARCO DETECTADOS:")
75 if results[0].boxes is not None:
76     for box in results[0].boxes:
77         cls_id = int(box.cls)
78         conf = float(box.conf)
79         print(f"→ {model.names[cls_id]} (confianza: {conf:.2%})")
80 else:
81     print(" No detectó barcos (normal en foto sin barcos)")
82
83 results[0].show()
84 %matplotlib inline
85 import glob
86 import random
87 import matplotlib.pyplot as plt
88 from ultralytics import YOLO
89
90 # 1. Cargar modelos
91 print("⌚ Cargando modelos...")
92 model_base = YOLO('yolo11n.pt')
93 model_trained = YOLO('runs/detect/seaships_yolo11n/weights/best.pt')
94
95 # 2. Tomar 3 imágenes al azar del test set (bajamos a 30 para que no
96 consuma
97 tanta RAM)
98 image_paths = glob.glob('test/images/*.jpg')
99 sample_images = random.sample(image_paths, min(30, len(image_paths)))
100
101 print("🖼️ Generando comparativa visual...")
102 for img_path in sample_images:
103     # Predicciones
104     res_base = model_base.predict(img_path, verbose=False)
105     res_trained = model_trained.predict(img_path, verbose=False)
106
107     # Obtener imágenes con las cajas dibujadas
108     img_base_plotted = res_base[0].plot()[..., :-1]
109     img_trained_plotted = res_trained[0].plot()[..., :-1]
110
```

```

111     # Dibujar la comparativa
112     fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))
113
114     ax1.imshow(img_base_plotted)
115     ax1.set_title("ANTES: Modelo BASE (General)", fontsize=14)
116     ax1.axis('off')
117
118     ax2.imshow(img_trained_plotted)
119     ax2.set_title("DESPUES: TU MODELO (Experto)", fontsize=14)
120     ax2.axis('off')
121
122     plt.tight_layout()
123     # Forzar que se muestre en Jupyter
124     display(fig)
125     # Limpiar memoria de la figura
126     plt.close(fig)
127
128     from ultralytics import YOLO
129     import glob
130     import os
131     import matplotlib.pyplot as plt
132     import torch
133
134     # --- LIMPIAR LA MEMORIA DE LA GPU ANTES DE EMPEZAR ---
135     if torch.cuda.is_available():
136         torch.cuda.empty_cache()
137         print("🧹 Memoria de la GPU liberada.")
138
139     # 1. Rutas absolutas al dataset ORIGINAL (SeaShips)
140     base_seaships = "/home/jrosria/SeaShips.v1-seaships.yolov11"
141
142     # Detección automática de la carpeta
143     if os.path.exists(f"{base_seaships}/valid/images"):
144         split = "valid"
145     elif os.path.exists(f"{base_seaships}/val/images"):
146         split = "val"
147     else:
148         split = "test"
149
150     img_dir = f"{base_seaships}/{split}/images"
151     label_dir = f"{base_seaships}/{split}/labels"
152     ruta_tu_modelo =
153     f"{base_seaships}/runs/detect/seaships_yolo11n/weights/best.pt"
154
155     archivos_img = glob.glob(os.path.join(img_dir, "*.*"))
156     archivos_txt = glob.glob(os.path.join(label_dir, "*.txt"))
157
158     print(f"📁 Procesando {len(archivos_img)} imágenes de {split}...")
159
160     if not archivos_img:
161         print("❌ ERROR: No se encontraron imágenes.")
162     else:
163         model_base = YOLO("yolo11n.pt")
164         model_entrenado = YOLO(ruta_tu_modelo)
165
166         barcos_reales = 0
167         for txt in archivos_txt:

```

```

168         with open(txt, 'r') as f:
169             barcos_reales += len(f.readlines())
170
171     barcos_base = 0
172     barcos_entrenado = 0
173
174     # --- PROCESAMIENTO EN LOTES (CHUNKS) PARA NO SATURAR LA RAM ---
175     batch_size = 32 # Procesar de 32 en 32 imágenes
176
177     print("🔍 Analizando con el modelo original en lotes...")
178     for i in range(0, len(archivos_img), batch_size):
179         lote = archivos_img[i:i + batch_size]
180         res_base = model_base.predict(source=lote, classes=[8],
181 conf=0.65, verbose=False)
182         for r in res_base:
183             barcos_base += len(r.bboxes)
184
185     # Liberamos memoria de nuevo antes del siguiente modelo
186     if torch.cuda.is_available():
187         torch.cuda.empty_cache()
188
189     print("🔍 Analizando con tu modelo entrenado en lotes...")
190     for i in range(0, len(archivos_img), batch_size):
191         lote = archivos_img[i:i + batch_size]
192         res_entrenado = model_entrenado.predict(source=lote,
193 conf=0.65, verbose=False)
194         for r in res_entrenado:
195             barcos_entrenado += len(r.bboxes)
196
197     print("\n--- RESULTADOS DEL MATCH (DATASET SEASHIPS) ---")
198     print(f"Total de barcos reales en las etiquetas:
199 {barcos_reales}")
200     print(f"Encontrados por YOLO Base: {barcos_base}")
201     print(f"Encontrados por Modelo Entrenado: {barcos_entrenado}")
202
203     # 5. Dibujar Gráfico
204     labels = ['YOLO Base (Fábrica)', 'YOLO Entrenado (Tuyo)']
205     detectados = [barcos_base, barcos_entrenado]
206
207     fig, ax = plt.subplots(figsize=(9, 6))
208     bars = ax.bar(labels, detectados, color=['#ff9999', '#66b3ff'],
209 edgecolor='black', width=0.5)
210
211     ax.axhline(y=barcos_reales, color='red', linestyle='--',
212 linewidth=2, label=f"Total de Barcos Reales ({barcos_reales})")
213
214     ax.set_ylabel("Cantidad de Barcos Detectados", fontsize=12,
215 fontweight='bold')
216     ax.set_title("Comparativa de Detección en Dataset Original
217 (SeaShips)", fontsize=14, fontweight='bold', pad=20)
218
219     max_y = max(barcos_reales, barcos_base, barcos_entrenado)
220     ax.set_ylim(0, max_y + (max_y * 0.2))
221     ax.legend(fontsize=11)
222     ax.grid(axis='y', linestyle='--', alpha=0.7)
223
224     for bar in bars:

```

```
225         yval = bar.get_height()
226         ax.text(bar.get_x() + bar.get_width()/2, yval + (max_y*0.02),
227               int(yval), ha='center', va='bottom', fontsize=12,
228               fontweight='bold')
229
230     plt.tight_layout()
231     plt.savefig('comparativa_seaships_original.png', dpi=300,
232               bbox_inches='tight')
233     plt.show()
```


ANEXO V: SCRIPT DETECTORCONFIG.CV

```
1 using UnityEngine;
2
3
4 public class DetectorConfig : MonoBehaviour
5 {
6     public static DetectorConfig Instance;
7
8     [Header("Ajustes de Detección")]
9     [Range(0f, 1f)] public float confidenceThreshold = 0.5f;
10    [Range(0f, 1f)] public float iouThreshold = 0.4f;
11
12    [Header("Rendimiento")]
13    [Range(1, 30)] public int detectEveryNFrames = 15;
14
15    [Header("Tracking (Seguimiento)")]
16    public float maxTrackingDistance = 150f;
17    public int maxLostFrames = 3;
18
19    void Awake()
20    {
21        if (Instance == null)
22        {
23            Instance = this;
24            DontDestroyOnLoad(gameObject); // persiste entre escenas
25        }
26        else
27        {
28            Destroy(gameObject);
29        }
30    }
31 }
```


ANEXO VI: SCRIPT PRUEBAS SERVIDOR

```

1  from ultralytics import YOLO
2  import glob
3  import os
4  import matplotlib.pyplot as plt
5
6  # 1. Rutas absolutas
7  img_dir = "/home/jrosria/dataset_prueba_externo/valid/images"
8  label_dir = "/home/jrosria/dataset_prueba_externo/valid/labels"
9
10 ruta_tu_modelo = "/home/jrosria/SeaShips.v1-
11 seaships.yolov11/runs/detect/seaships_yolo11n/weights/best.pt"
12
13 # Obtener imágenes y etiquetas
14 archivos_img = glob.glob(os.path.join(img_dir, "*.jpg"))
15 archivos_txt = glob.glob(os.path.join(label_dir, "*.txt"))
16
17 print(f" Procesando {len(archivos_img)} imágenes y {len(archivos_txt)}
18 etiquetas...")
19
20 # 2. Cargar modelos
21 print("Cargando modelo YOLO11n original (COCO)...")
22 model_base = YOLO("yolo11n.pt") # Descarga automática del original
23
24 print("Cargando TU modelo entrenado...")
25 model_entrenado = YOLO(ruta_tu_modelo)
26
27 # 3. Contar barcos reales (etiquetas físicas)
28 barcos_reales = 0
29 for txt in archivos_txt:
30     with open(txt, 'r') as f:
31         barcos_reales += len(f.readlines())
32
33 # 4. Hacer predicciones
34 barcos_base = 0
35 barcos_entrenado = 0
36
37 print("🔍 Analizando con el modelo original (Solo clase 8 = barco)...")
38 res_base = model_base.predict(source=archivos_img, classes=[8], conf=0.25,
39 verbose=False)
40 for r in res_base:
41     barcos_base += len(r.bboxes)
42
43 print(" Analizando con tu modelo entrenado...")
44 res_entrenado = model_entrenado.predict(source=archivos_img, conf=0.25,
45 verbose=False)
46 for r in res_entrenado:
47     barcos_entrenado += len(r.bboxes)
48
49 print("\n--- RESULTADOS DEL MATCH ---")
50 print(f"Total de barcos reales en las etiquetas: {barcos_reales}")
51 print(f"Encontrados por YOLO Base: {barcos_base}")
52 print(f"Encontrados por TU Modelo: {barcos_entrenado}")
53
54 # 5. Dibujar Gráfico
55 labels = ['YOLO Base (Fábrica)', 'YOLO Entrenado (Tuyo)']

```

```
56 detectados = [barcos_base, barcos_entrenado]
57
58 fig, ax = plt.subplots(figsize=(9, 6))
59 bars = ax.bar(labels, detectados, color=['#ff9999', '#66b3ff'],
60 edgecolor='black', width=0.5)
61
62 # Línea meta (Total real)
63 ax.axhline(y=barcos_reales, color='red', linestyle='--', linewidth=2,
64 label=f'Total de Barcos Reales ({barcos_reales})')
65
66 ax.set_ylabel('Cantidad de Barcos Detectados', fontsize=12,
67 fontweight='bold')
68 ax.set_title('Comparativa de Sensibilidad: Detección en Dataset Externo',
69 fontsize=14, fontweight='bold', pad=20)
70
71 max_y = max(barcos_reales, barcos_base, barcos_entrenado)
72 ax.set_ylim(0, max_y + (max_y * 0.2))
73 ax.legend(fontsize=11)
74 ax.grid(axis='y', linestyle='--', alpha=0.7)
75
76 # Etiquetas numéricas
77 for bar in bars:
78     yval = bar.get_height()
79     ax.text(bar.get_x() + bar.get_width()/2, yval + (max_y*0.02),
80             int(yval), ha='center', va='bottom', fontsize=12,
81 fontweight='bold')
82
83 plt.tight_layout()
84 plt.savefig('comparativa_detectados.png', dpi=300, bbox_inches='tight')
85 plt.show()
86 from ultralytics import YOLO
87 import glob
88 import os
89 import matplotlib.pyplot as plt
90
91 # 1. Rutas absolutas
92 img_dir = "/home/jrosria/dataset_prueba_externo/valid/images"
93 label_dir = "/home/jrosria/dataset_prueba_externo/valid/labels"
94 ruta_tu_modelo = "/home/jrosria/SeaShips.v1-
95 seaships.yolov11/runs/detect/seaships_yolov11n/weights/best.pt"
96
97 # Obtener imágenes y etiquetas
98 archivos_img = glob.glob(os.path.join(img_dir, "*.*"))
99 archivos_txt = glob.glob(os.path.join(label_dir, "*.txt"))
100
101 # 2. Cargar modelos
102 model_base = YOLO("yolov11n.pt")
103 model_entrenado = YOLO(ruta_tu_modelo)
104
105 # 3. Contar barcos reales (etiquetas físicas)
106 barcos_reales = 0
107 for txt in archivos_txt:
108     with open(txt, 'r') as f:
109         barcos_reales += len(f.readlines())
110
111 # 4. Hacer predicciones CON UMBRAL ALTO (conf=0.65) para evitar
112 overprediction
```

```

113 barcos_base = 0
114 barcos_entrenado = 0
115
116 print("🔍 Analizando con el modelo original (Umbral 65%)...")
117 # Se sube conf a 0.65 para eliminar fragmentaciones
118 res_base = model_base.predict(source=archivos_img, classes=[8], conf=0.65,
119 verbose=False)
120 for r in res_base:
121     barcos_base += len(r.bboxes)
122
123 print("🔍 Analizando con tu modelo entrenado (Umbral 65%)...")
124 res_entrenado = model_entrenado.predict(source=archivos_img, conf=0.65,
125 verbose=False)
126 for r in res_entrenado:
127     barcos_entrenado += len(r.bboxes)
128
129 print("\n--- RESULTADOS DEL MATCH (CALIBRADO) ---")
130 print(f"Total de barcos reales en las etiquetas: {barcos_reales}")
131 print(f"Encontrados por YOLO Base: {barcos_base}")
132 print(f"Encontrados por TU Modelo: {barcos_entrenado}")
133
134 # 5. Dibujar Gráfico
135 labels = ['YOLO Base (Fábrica)', 'YOLO Entrenado (Tuyo)']
136 detectados = [barcos_base, barcos_entrenado]
137
138 fig, ax = plt.subplots(figsize=(9, 6))
139 bars = ax.bar(labels, detectados, color=['#ff9999', '#66b3ff'],
140 edgecolor='black', width=0.5)
141
142 # Línea meta (Total real)
143 ax.axhline(y=barcos_reales, color='red', linestyle='--', linewidth=2,
144 label=f'Total de Barcos Reales ({barcos_reales})')
145
146 ax.set_ylabel('Cantidad de Barcos Detectados', fontsize=12,
147 fontweight='bold')
148 ax.set_title('Comparativa de Detección (Calibrada al 65% de Confianza)',
149 fontsize=14, fontweight='bold', pad=20)
150
151 # Ajustar límite superior
152 max_y = max(barcos_reales, barcos_base, barcos_entrenado)
153 ax.set_ylim(0, max_y + (max_y * 0.2))
154 ax.legend(fontsize=11)
155 ax.grid(axis='y', linestyle='--', alpha=0.7)
156
157 # Etiquetas numéricas
158 for bar in bars:
159     yval = bar.get_height()
160     ax.text(bar.get_x() + bar.get_width()/2, yval + (max_y*0.02),
161             int(yval), ha='center', va='bottom', fontsize=12,
162 fontweight='bold')
163
164 plt.tight_layout()
165 plt.savefig('comparativa_detectados_calibrada.png', dpi=300,
166 bbox_inches='tight')
167 plt.show()

```


ANEXO VII: USO DE INTELIGENCIA ARTIFICIAL

En el desarrollo del presente TFG se ha empleado la inteligencia artificial como herramienta de apoyo en dos ámbitos: por un lado, en la generación de imágenes utilizadas para el diseño y desarrollo de la interfaz visual de la aplicación y, por otro, como asistencia en la programación, facilitando la redacción, revisión y depuración del código necesario para la implementación del sistema. Este uso ha permitido agilizar tareas técnicas, manteniendo en todo momento el criterio en la toma de decisiones, la integración de los distintos elementos y la validación final del funcionamiento de la aplicación desarrollada.

Se muestran a continuación los *prompts* para la generación de las imágenes:

1. Tuerca slider (Figura 3-15). Prompt empleado: “Pixel art de una tuerca hexagonal metálica gigante, estilo cyberpunk o ciencia ficción. La tuerca tiene aspecto desgastado y ligeramente oxidado, con pequeños remaches en la cara frontal. En el lado izquierdo, tiene varios cables de colores (rojo, azul, amarillo y gris) enrollados y sujetos con pequeñas abrazaderas metálicas. Vista en perspectiva de tres cuartos, iluminación estilizada, fondo blanco limpio”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 20 febrero de 2026. Enlace: <https://gemini.google.com/>
2. Fondo slider (Figura 3-15). Prompt empleado: “Pixel art de un muelle metálico industrial dentro de una carcasa o marco rectangular. Estilo cyberpunk o ciencia ficción retro. El marco es de color acero azulado, con un aspecto desgastado, pequeñas marcas de óxido y remaches en los extremos. En el interior, un resorte helicoidal denso y oscuro con iluminación que resalta la textura del metal. Perspectiva frontal plana 2D, iluminación estilizada, fondo blanco limpio”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 20 febrero de 2026. Enlace: <https://gemini.google.com/>
3. Letrero menú (Figura 3-16). Prompt empleado: “Un letrero futurista estilo industrial con la palabra "MENU" en letras grandes 3D de metal rojo, montadas sobre un panel oscuro de acero, rodeado de detalles mecánicos como tornillos, medidores y cables enrollados alrededor de las letras, estilo cyberpunk, texturas desgastadas con rayones y suciedad, iluminación dramática con alto contraste, sombreado detallado, estilo interfaz de videojuego, composición simétrica, ultra detallado, calidad 4K”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 20 febrero de 2026. Enlace: <https://gemini.google.com/>
4. Botón X (Figura 3-14). Prompt empleado: “Icono de botón estilo pixel art detallado, una gran X roja metálica en 3D en el centro de un panel industrial oscuro, cables negros enrollados alrededor de la X, luces rojas pequeñas en las esquinas, medidores y placas mecánicas, textura con desgaste y suciedad, estilo videojuego retro sci-fi, sombreado pixel perfecto, bordes definidos, fondo transparente, alta resolución”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 20 febrero de 2026. Enlace: <https://gemini.google.com/>
5. Botón EXIT (Figura 3-14). Prompt empleado: “Un letrero futurista estilo industrial con la palabra "EXIT" en letras grandes 3D de metal rojo, montadas sobre un panel oscuro de acero, rodeado de detalles mecánicos como tornillos, medidores y cables enrollados alrededor de las letras, estilo cyberpunk, texturas desgastadas con rayones y suciedad, iluminación dramática con alto contraste, sombreado detallado, estilo interfaz de videojuego, composición simétrica, ultra detallado, calidad 4K”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 20 febrero de 2026. Enlace: <https://gemini.google.com/>
6. Fondo menú (Figura 3-16). Prompt empleado: “Pixel art de un pasillo oscuro y angosto en el interior de una nave espacial o instalación industrial de ciencia ficción. Las paredes son

- metálicas de color azul grisáceo oscuro, llenas de paneles eléctricos, tuberías gruesas en el techo, cajas de interruptores con pequeñas luces LED rojas y amarillas, y escotillas pesadas. El suelo es de rejilla metálica. Iluminación sombría y atmosférica con sombras duras, estilo retro cyberpunk o 16-bit. Perspectiva con punto de fuga central creando profundidad”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 18 febrero de 2026. Enlace: <https://gemini.google.com/>
7. Marco menú(Figura 3-16) . Prompt empleado: “Pixel art de un marco rectangular metálico grueso para una interfaz de usuario de videojuego, estilo ciencia ficción industrial. El marco es de paneles de acero azulado oscuro ensamblados con pequeños remaches. En la esquina superior izquierda salen varios cables de colores (rojo, amarillo, azul oscuro). Tiene pequeños detalles mecánicos incrustados: un pequeño panel de luces en la parte superior central y un medidor analógico amarillo con un interruptor en la esquina inferior derecha. El interior del marco está completamente en blanco y vacío. Vista frontal plana 2D, sombreado nítido de 16-bit. Perspectiva con punto de fuga central creando profundidad”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 18 febrero de 2026. Enlace: <https://gemini.google.com/>
 8. Botón menú (Figura 3-11). Prompt empleado: “Pixel art de un botón de interfaz de usuario cuadrado, estilo industrial de ciencia ficción. El botón está hecho de paneles de metal gris acero unidos con pequeños remaches. En el centro tiene tres barras horizontales en relieve que funcionan como agarre o textura. Tiene un borde exterior biselado; el borde inferior es de un color rojo brillante con sombreado oscuro, dando un efecto 3D de profundidad de botón pulsable. Pequeños detalles amarillos en las esquinas superiores. Vista frontal 2D plana, sombreado clásico de 16-bit, fondo blanco limpio”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 18 febrero de 2026. Enlace: <https://gemini.google.com/>
 9. Menú inicial (Figura 3-18). Prompt empleado: “Ilustración digital 2D en estilo 'línea clara' de un puerto militar y academia naval en un día soleado de verano. En primer plano, varios cañones costeros grises y anclas de barco sobre un paseo adoquinado. Un grupo de marineros en uniforme blanco con gorras están observando y tomando notas. Al fondo, edificios de la academia naval con arquitectura clásica, techos de tejas naranjas y arcos. En el agua azul brillante, se ven barcos de guerra grises atracados. El cielo es azul claro con nubes blancas esponjosas y banderines náuticos coloridos colgados en diagonal. Colores vivos y vibrantes, trazos limpios, estilo de animación moderna”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 24 febrero de 2026. Enlace: <https://gemini.google.com/>
 10. Altavoz (Figura 3-18). Prompt empleado: “Pixel art de un altavoz o bocina estilo industrial de ciencia ficción, visto de perfil apuntando hacia la derecha. La bocina está construida con paneles metálicos grises con pequeños remaches. El borde exterior del cono es de color rojo brillante con un fino contorno amarillo. A la derecha de la bocina salen tres ondas de sonido curvas, coloreadas en rojo con contorno amarillo y sombra negra. Estilo retro de 16-bit, contornos negros gruesos (pixel art outline), fondo blanco limpio”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 24 febrero de 2026. Enlace: <https://gemini.google.com/>
 11. Altavoz mute (Figura 3-18). Prompt empleado: “Pixel art de un altavoz o bocina estilo industrial de ciencia ficción, visto de perfil apuntando hacia la derecha. La bocina está construida con paneles metálicos grises con pequeños remaches y un borde rojo en el cono. Sobre la parte delantera de la bocina hay una gran 'X' roja gruesa tachándola, indicando silencio o 'mute'. La 'X' tiene sombreado para darle relieve y volumen. Estilo retro de 16-bit, contornos negros gruesos (pixel art outline), fondo blanco limpio”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 24 febrero de 2026. Enlace: <https://gemini.google.com/>

12. Letrero inicio (Figura 3-18). Prompt empleado: “Pixel art del título de un videojuego que dice 'SHIP DETECTOR' en letras mayúsculas gruesas. Estilo industrial de ciencia ficción. Las letras están hechas de paneles metálicos grises con pequeños remaches plateados. La parte inferior de cada letra tiene una franja roja gruesa y un borde inferior azul oscuro, dando un efecto 3D de bloque. Sobre algunas de las letras hay pequeños cañones navales y de artillería incrustados como parte del diseño. Contornos negros gruesos (pixel art outline), fondo blanco limpio”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 24 febrero de 2026. Enlace: <https://gemini.google.com/>
13. Botón start(Figura 3-18) . Prompt empleado: “Pixel art del título de un videojuego que dice 'START' en letras mayúsculas gruesas. Estilo industrial de ciencia ficción. Las letras están hechas de paneles metálicos grises con pequeños remaches plateados. La parte inferior de cada letra tiene una franja roja gruesa y un borde inferior azul oscuro, dando un efecto 3D de bloque. Contornos negros gruesos (pixel art outline), fondo blanco limpio”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 24 febrero de 2026. Enlace: <https://gemini.google.com/>
14. Marcador(Figura 3-10). Prompt empleado: “Pixel art de un marco rectangular metálico alargado para interfaz de usuario, estilo industrial de ciencia ficción. El marco es sencillo, hecho de paneles de acero gris claro con pequeños remaches. El borde exterior es de color azul marino oscuro y el borde inferior tiene una gruesa línea roja brillante con dos pequeños acentos amarillos cerca de las esquinas inferiores. El interior del marco está completamente en blanco y vacío. Vista frontal plana 2D, sombreado clásico de 16-bit, fondo blanco limpio”. Herramienta utilizada: Gemini. Versión: Gemini 3.1 Pro Thinking. Compañía autora: Google. Fecha de la consulta: 24 febrero de 2026. Enlace: <https://gemini.google.com/>

ANEXO VIII: IMPLICACIONES SOCIALES, ECONÓMICAS, Y AMBIENTALES

El empleo de tecnologías de vigilancia marítima basadas en visión artificial genera un impacto tanto a nivel económico como operativo, al reforzar la seguridad en entornos portuarios y zonas costeras. Considerando la alta densidad de tráfico que transita cada año las rutas marítimas comerciales próximas a España, la capacidad de detectar y clasificar embarcaciones en tiempo real proporciona una ventaja en el conocimiento del entorno. En este contexto, los modelos de detección de objetos, como YOLO, implementados en aplicaciones usando Unity y Sentis para la gestión de los modelos de IA y adaptados a dispositivos móviles, permite mejorar la identificación visual en el ámbito marítimo.

Este tipo de aplicaciones facilita la anticipación de riesgos, disminuye la probabilidad de clasificaciones erróneas y mejora la eficiencia de las maniobras portuarias. Como resultado, se logra una reducción significativa de los costes derivados de accidentes, retrasos logísticos y daños materiales que pueden provocar impactos medioambientales en zonas costeras y portuarias. En este sentido, el uso de estas aplicaciones contribuye a mitigar estos riesgos, fomentando una navegación más segura, eficiente y respetuosa con el entorno marino.

ANEXO IX: REFLEXIONES ÉTICAS Y SOCIALES

El desarrollo de sistemas de detección de embarcaciones basados en visión artificial posee una gran versatilidad, con aplicaciones tanto en el ámbito civil como en el militar. La detección y clasificación, funciones intrínsecas de la vigilancia marítima, mediante modelos de detección de objetos crea implicaciones éticas relacionadas con la responsabilidad y sus posibles usos. En este sentido, el principio de responsabilidad formulado por Hans Jonas establece la necesidad de anticipar las consecuencias y actuar de forma prudente ante posibles problemas, especialmente cuando estos pueden derivar en una reducción de la privacidad. Por ello, es necesario crear mecanismos de supervisión que garanticen su uso conforme a principios éticos y legales.

Desde una perspectiva social, la aplicación desarrollada aporta un valor a la seguridad marítima. El análisis visual y clasificación del tráfico naval permite mejorar la calidad de la información disponible para la toma de decisiones. En este contexto, la aportación de estos sistemas puede interpretarse como concepto de racionalidad limitada desarrollado por Herbert A. Simon, según el cual los individuos operan bajo restricciones cognitivas que limitan su capacidad para procesar información compleja. Así, la inteligencia artificial actúa como un mecanismo de apoyo que amplía las capacidades humanas en entornos altamente dinámicos y saturados de datos, contribuyendo a la protección de vidas humanas y a la eficiencia de las operaciones portuarias y de patrullaje.