ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Máster

Aplicación de Deep Learning al análisis de especies en fondos marinos

(Deep Learning application to species analysis in seabed)

Para acceder al Título de

Máster Universitario en Ingeniería de Telecomunicación

Autor: Sergio Sierra Menéndez

Octubre -2020

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACION

MASTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE MASTER

Realizado por: Sergio Sierra Menéndez
Director del TFM: Elena Prado Ortega
Título: "Aplicación de Deep Learning al análisis de especies en fondos marinos"
Title: "Deep Learning application to species analysis in seabed"

Presentado a examen el día:

para acceder al Título de

MASTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Composición del Tribunal:

Presidente (Apellidos, Nombre): Casanueva López, Alicia

Secretario (Apellidos, Nombre): Conde Portilla, Olga

Vocal (Apellidos, Nombre): Basterrechea Verdeja, José

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente Fdo.: El Secretario

Fdo.: El Vocal Fdo.: El Director del TFM

Vº Bº del Subdirector Trabajo Fin de Máster Nº

"Trabajo realizado en parte con los medios aportados por los proyectos TEC2016-76021-C2-2-R, PID2019-107270RB-C21, DeepRamp (Pleamar 2019, FB) y VirtualMar 2470S (PPNN 2017)"

Agradecimientos

En este apartado quiero agradecer a todas las personas que han contribuido de alguna manera a la realización de este trabajo.

En primer lugar, quiero agradecer a Adolfo, su paciencia, dedicación y confianza en mí. No habría sido posible llevar a cabo este proyecto sin ti.

A Francisco Sánchez y Elena Prado del instituto Español de Oceanografía (IEO) por la información suministrada y la confianza depositada.

A mi familia por su apoyo, en especial a mi madre.

Finalmente, a Sandra y a mis amigos por esos momentos de desconexión tan necesarios.

Índice general

Índice de figuras

Índice de tablas

Resumen

Este trabajo se basa en la aplicación de técnicas de procesado de imágenes con herramientas ***Deep learning***, como la segmentación de imágenes y la detección de objetos sobre el lenguaje de programación ***Python***.

Se trata de conseguir la clasificación y análisis del coral del fondo marino, clasificándolo entre coral vivo y muerto, obteniéndose así el porcentaje de área superviviente de la especie.

Igualmente, se trabajará con redes de detección de objetos, que clasificarán especies marinas abundantes en el entorno marino de los cañones de Avilés.

Todo esto, sustentado gracias a las imágenes obtenidas por los investigadores del ***Instituto Español de Oceanografía*** (IEO).

Abstract

This project is based on the application of image processing techniques with **Deep learning** such as the segmentation of images and the detection of objects on the programming language **Python**.

The main idea is based on achieving the classification and analysis of the seafloor, classifying it between living and dead coral, obtaining the percentage of the species' surviving area.

The second point of the project will be done with object detection networks, which will classify abundant marine species in the marine environment of the Avilés canyons.

All this, based on the images obtained by the researchers of the Spanish **Institute of Oceanography**.

Capítulo 1

Introducción

El ***Deep learning*** o aprendizaje profundo es un conjunto de algoritmos de aprendizaje automático que busca modelar información en datos usando arquitecturas computacionales que admiten transformaciones no lineales múltiples e iterativas de datos expresados en forma matricial o tensorial.

A día de hoy, los datos y la información son oro. Por este motivo, ha habido un gran desarrollo y un mayor interés por parte de las empresas y público en general por esta herramienta en los últimos años.

Así mismo, se ha postulado como una de las mejores opciones respecto a tecnologías hermanas como el ***Machine Learning*** , esto se debe a que trabaja mejor con grandes cantidades de datos para el procesado.

Durante el proyecto, se aplicarán técnicas de procesado de imágenes con ***Deep learning***, tratando de conseguir la clasificación y análisis automático de especies marinas en fondos de gran profundidad, en este caso, la especie a tratar es el coral, siendo el objetivo clasificarlo entre vivo y muerto a partir de imágenes obtenidas con **vehículos remotamente operados** (ROV) por los investigadores del **Instituto Español de Oceanografía** (IEO).

En esta ocasión, se va a hacer uso de redes CNN pre-entrenadas a través de **Keras+ Tensorflow** y **Pytorch**, ambas son bibliotecas especializadas en aplicaciones de aprendizaje automático y redes neuronales profundas. En la actualidad, hay muchos proyectos que basan su funcionamiento en estas bibliotecas, siendo sobretodo **PyTorch** una de la que más ha aumentado su popularidad en los últimos años. Sin embargo, hay varías librerías con un buen nivel de desarrollo, que se podrían utilizar en un futuro comprobando su potencial.

1.1. Motivación

Hasta hace unos años, que un robot fuese capaz de asemejarse al comportamiento humano era solo una quimera, aparecía en películas de ciencia ficción y en los sueños de las personas

más optimistas.

Sin embargo, debido al impulso generado por la revolución del ***Big Data***, en la actualidad, cada vez son más las empresas que buscan la implementación de técnicas para la mejora de la producción gracias a la **Inteligencia Artificial**.

El ***Deep Learning***[17], es un subconjunto dentro del campo del ***Machine Learning***, su objetivo es emular el comportamiento del ser humano, conllevando esto múltiples aplicaciones en el mundo del ***Big Data*** y el **Internet de las cosas**.

Entre las múltiples aplicaciones del ***Deep Learning*** podemos encontrar:

- Traductores inteligentes.
- Lenguaje hablado y escrito.
- Reconocimiento de voz.
- Interpretación de la semántica.

Para este proyecto nos centraremos en esta última. Siendo este referente a la identificación de rostros y objetos en vídeos y fotografías.

1.2. Objetivos

Este trabajo consta de dos temas/objetivos principales, ambos relacionados con el estudio y el seguimiento de hábitats marinos con la intención de conservarlos.

El primer tema consiste en abordar el estudio del hábitat marino y obtener un seguimiento para la conservación de estos hábitats diferenciando entre coral vivo/muerto.

Figura 1.1: Coral vivo vs coral muerto. Imagen recuperada de **La Vanguardia** .

Para ello, se realizará de forma automática la detección de coral tanto vivo como muerto, comprobando un conjunto de vídeos e imágenes del fondo marino, esto se llevará a cabo con una de las herramientas que nos ofrece el ***Deep learning***, la segmentación de imagen. Gracias a esta herramienta seremos capaces de estimar a qué clase pertenece cada píxel de una imagen, facilitando de esta forma el cálculo del área sobreviviente.

Esto podrá servir, entre otras cosas, para estudiar la densidad de esta especie viva, en cada intervalo de tiempo, comprobando así el porcentaje sobreviviente y su área total.

El segundo tema trata la detección y localización de especies marinas que habitan en la zona de los cañones de Avilés. Estas especies son las siguientes:

- **Artemisina (Artemisina transiens):** se trata de una esponja marina de forma redondeada y reducido tamaño.
- **Phakellia (Phakellia ventilabrum)**: al igual que la artemisina se trata de una esponja marina, su tamaño es mayor que el de esta y suele tener un tono blanquecino.
- **Dendrophyllia (D. cornigera)**: esta especie marina a diferencia de los anteriores se trata de uno de los múltiples tipos de coral existentes. Su color es amarillento.

En este punto también se hará uso de herramientas de ***Deep learning***, sin embargo, estas serán enfocadas a la detección y localización de cada objeto correspondiente a las tres clases anteriores y no a la clasificación a nivel de píxel del primer tema.

Actualmente, esta labor se realiza de forma manual por un biólogo que visualiza los vídeos, una tarea muy tediosa o que simplemente no puede hacerse por falta de personal. Por ello, se va a diseñar e implementar una **red neuronal artificial** (RNA) de tipo ***Deep Learning***.

Con la ayuda de una serie de datos proporcionados por el **Instituto Español de Oceanografía** (IEO), se creará en primer lugar el set de datos para el entrenamiento, se llevará a cabo el entrenamiento, se comprobará cuál de los métodos es más funcional y por último, el lector podrá observar algunos ejemplos de inferencia y métricas obtenidas por cada uno de los modelos.

Capítulo 2

Fundamentos teóricos

2.1. Medios para la toma de imágenes y vídeos

El ***Deep Learning*** se sustenta en sets de datos, ya sea para el entrenamiento, para la validación o para testear el funcionamiento del modelo. En esta ocasión, los datos provienen de una colección de vídeos e imágenes aportados por el **IEO**[2]. Estos fotogramas son grabados con dos tipos de submarinos principalmente:

2.1.1. Submarino Politolana

El primero de ellos es el **Politolana**, se trata de un submarino tipo trineo, este es arrastrado por un barco con un cable haciendo así que se vaya desplazando por el fondo marino, lleva incorporada una cámara de vídeo de alta definición (**1920 x 1080 pixels**), una cámara fotográfica de 24Mpx (Nikon D90) e iluminación mediante dos focos LED Sphere de última generación (**12600 lumens**), que graba el recorrido en todo momento.

Figura 2.1: Submarino Politolana [35]

2.1.2. Submarino Lander

Otro tipo de submarino utilizado es el denominado **Lander**, este, a diferencia del Politolana, es equipo fijo con trípode que reposa en el mismo punto durante largos periodos de tiempo, sacando fotografías del mismo punto constantemente hasta su extracción. Este tipo de información es útil para ver el desarrollo de una zona concreta con el paso del tiempo.

Figura 2.2: Submarino Lander [36]

2.2. Machine Learning

Tal y como se menciona en el blog de Indra[23], "***Machine Learning*** se describe a menudo como un tipo de técnicas de **Inteligencia Artificial** donde las computadoras aprenden a hacer algo sin ser programadas para ello. Por poner un ejemplo sencillo, se podría programar un ordenador para identificar a un animal como un gato escribiendo un código que indique al programa que elija "gato" cuando se ve una imagen concreta de un gato. Esto funcionaría si el único gato con el que tratase el programa es el de esa imagen, pero no lo haría si el programa tuviera que ver un montón de imágenes de diferentes animales, incluyendo una gran cantidad de gatos, y tuviera que identificar cuáles de ellas representan a un gato."

En este proyecto necesitamos que la herramienta detecte, no solo el coral existente, sino también las distintas variedades de coral que aparecen.

Por lo tanto, el método más apropiado para esta ocasión es el ***Deep Learning***.

2.3. Deep Learning

De nuevo en el blog de **Indra**[23] se define de la siguiente manera: "El ***Deep Learning*** lleva a cabo el proceso de *Machine Learning* usando una red neuronal artificial que se compone de un número de niveles jerárquicos. En el nivel inicial de la jerarquía la red aprende algo simple y luego envía esta información al siguiente nivel. El siguiente nivel toma esta información sencilla, la combina, compone una información algo un poco más compleja, y se lo pasa al tercer nivel, y así sucesivamente.

Continuando con el ejemplo del gato, el nivel inicial de una red de *Deep Learning* podría utilizar las diferencias entre las zonas claras y oscuras de una imagen para saber dónde están los bordes de la imagen. El nivel inicial pasa esta información al segundo nivel, que combina los bordes construyendo formas simples, como una línea diagonal o un ángulo recto. El tercer nivel combina las formas simples y obtiene objetos más complejos como óvalos o rectángulos. El siguiente nivel podría combinar los óvalos y rectángulos, formando barbas, patas o colas rudimentarias. El proceso continúa hasta que se alcanza el nivel superior en la jerarquía, en el cual la red aprende a identificar gatos."

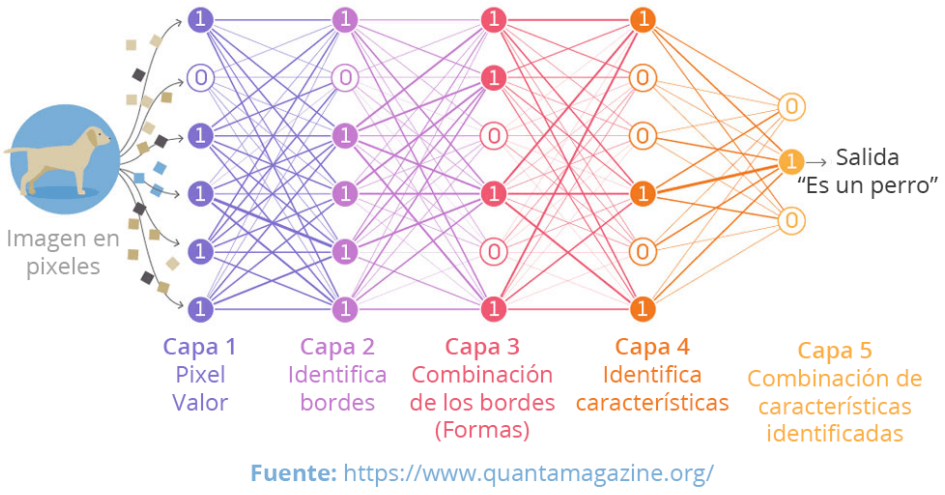


Figura 2.3: Deep learning por capas.

El ***Deep Learning*** se basa en mecanismos de aprendizaje automático, o ***Machine Learning***, llevándose a cabo un aprendizaje de principio a fin.

Estos métodos no son novedosos, sin embargo, su popularidad ha aumentado a lo largo de los últimos años, debido a sus diversas virtudes y ayudas a la sociedad como:

- Se ha demostrado que al utilizarse ***Deep Learning*** se pueden obtener resultados notablemente mejores que utilizando técnicas estándar de visión artificial.
- Conducción autónoma.

- Ayuda en cuanto a la mejor comprensión de enfermedades, mutaciones de enfermedades y terapias genéticas.
- Detección facial por parte de la policía.

En definitiva, beneficios que sin duda hacen más fácil el día a día a la sociedad y a los trabajadores.

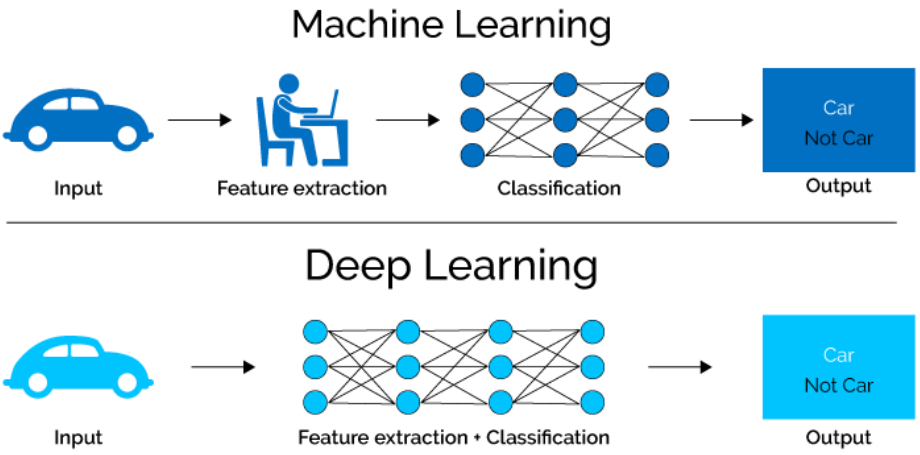


Figura 2.4: Machine Learning vs Deep Learning. *Imagen de* ***OpenWebinars***

2.3.1. Redes Neuronales Artificiales (RNA)

Las redes neuronales artificiales son una técnica de aprendizaje y procesamiento automático inspirada en el funcionamiento del cerebro humano. Podemos definir las redes neuronales artificiales[3] como un modelo computacional, paralelo, compuesto de unidades procesadoras conectadas entre sí.

Gracias a estas redes, se pueden lograr las siguientes ventajas:

- Son sistemas distribuidos no lineales: Una neurona es un elemento no lineal por lo que una interconexión de ellas (red neuronal) también será un dispositivo no lineal. Esta propiedad permitirá la simulación de sistemas no lineales y caóticos, que con sistemas clásicos sería imposible de realizar.
- Son sistemas tolerantes a fallos, una red neuronal, al ser un sistema distribuido, permite el fallo de algunos elementos individuales (neuronas) sin alterar significativamente la respuesta total del sistema.
- Establecen relaciones no lineales entre datos mediante relaciones complejas.

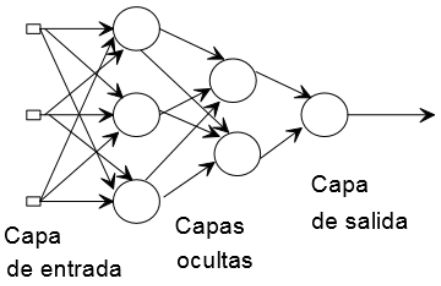


Figura 2.5: Red neuronal artificial.

2.3.2. Elementos básicos de una red neuronal artificial.

En toda red neuronal artificial se encuentran cuatro elementos básicos. Estos son los siguientes:

- Un conjunto de conexiones, pesos o sinapsis que determinan el comportamiento de la neurona. Estas conexiones pueden ser excitadoras (presentan un signo positivo), o inhibidoras (conexiones negativas).
- Un sumador que se encarga de sumar todas las entradas multiplicadas por las respectivas sinapsis.
- Una función de activación no lineal para limitar la amplitud de la salida de la neurona.
- Un umbral exterior que determina el umbral por encima del cual la neurona se activa.

2.4. Red neuronal convolucional

Las **redes neuronales convolucionales**[3] trabajan con pequeñas piezas de información, que son combinadas en las capas más profundas de la red. Cada capa tiene una función diferente, por poner un ejemplo fácil de entender, la primera capa trata de detectar los bordes estableciendo patrones de detección de bordes. Las capas posteriores tratan de combinarlos en formas más simples, finalmente combinándolos en patrones por cada característica de un objeto, como puede ser la iluminación, las escalas, etc.

Las últimas capas tratan de hacer coincidir la imagen de entrada, con todos los patrones de las capas anteriores determinando una predicción final como una suma ponderada de todos ellos.

De esta forma, las **redes neuronales convolucionales** consiguen modelar variaciones y comportamientos obteniendo predicciones muy precisas.

Capas de una red neuronal convolucional

Las CNNs[17] se conforman por varias etapas:

- En la primera etapa se produce la **extracción de características**, se extraen las características que poseen los datos de entrada (**píxeles**), formando los mapas de características (**feature maps**),que aumentan en unidades y disminuyen en tamaño tras el paso por las capas.

  Por ejemplo, si se busca caracterizar el color de una imagen se tendrían 3 etapas, una para el color rojo, otra para el color verde y otra para el azul.

- En la segunda etapa, se encuentran las llamadas **capas ocultas**. En ella están incluidas dos capas, la primera, es la **capa convolucional** y la segunda la **capa de pooling**, estas dos capas van en cascada y en algunos artículos son mencionadas como una sola capa (convolucional + pooling).

  Esta etapa se va ejecutando en un bucle hasta conseguir **mapas de características** de muy baja resolución. Siendo este el momento en el que se produce el paso a la etapa de clasificación.

- En las **capas de salida** se une la última convolución con una (ó más) capas de neuronas ocultas, que buscan clasificar por ejemplo, si lo que le hemos pasado a la entrada es un perro o un gato.

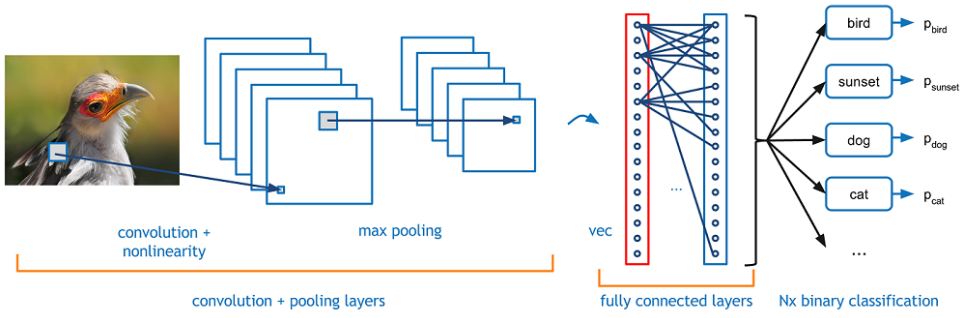


Figura 2.6: Ejemplo de Red Neuronal Convolucional de clasificación de imagen [10]

2.5. Redes de segmentación de imagen

Tal y como se ve en la **Figura 2.7** o en la **Figura 1.1**, la diferencia principal entre ambas imágenes (coral vivo y coral muerto) es el color de este. Por lo tanto, parece claro que se debe utilizar una red que le dé cierta importancia a este tipo de aspecto.

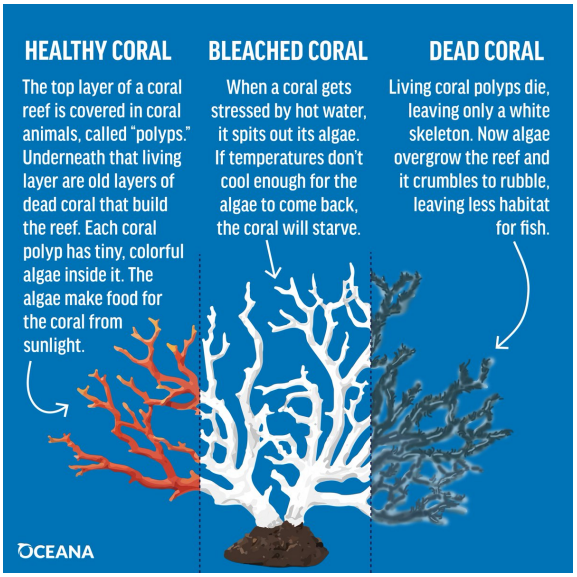


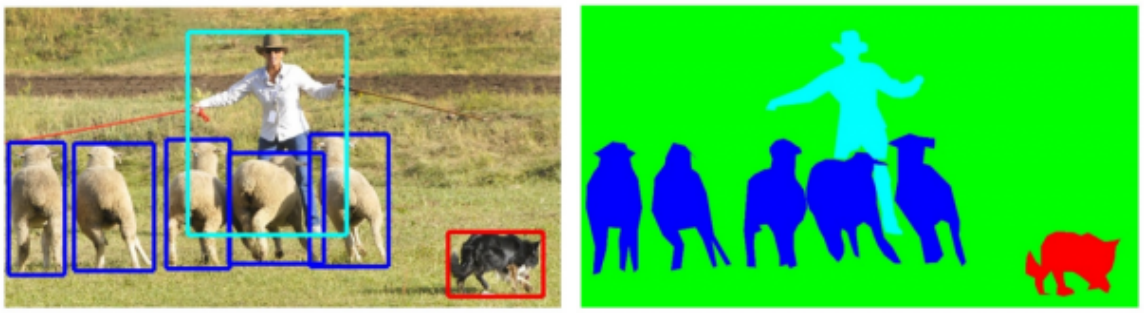
Figura 2.7: Definición de coral vivo y muerto. *Imagen de* ***Oceana***.

A lo largo del proyecto se va a considerar como coral vivo el color rojizo de la imagen de la parte izquierda de la **Figura 2.7** y el color central (blanco), por otro lado, el color de la derecha se tratará como coral muerto. Este es un color parecido al de los sedimentos que se encuentran en el fondo marino, por lo que por suerte no parece en principio demasiado difícil de diferenciar para el posterior etiquetado del ***dataset.***

Para este tipo de casos, se suele utilizar **redes de segmentación de imagen**, ya que estas se basan en la detección por la diferencia de **niveles de gris**, es decir, el color. Hay dos métodos básicos de detección siendo estos los siguientes:

- **Discontinuidades del nivel de gris.** Este tipo de método consiste en delimitar la imagen a partir de los cambios producidos en los niveles de gris entre diferentes píxeles. Las técnicas que utilizan las discontinuidades como base son la detección de líneas, de bordes...
- **Similitud de niveles de gris.** En este método se detectan las divisiones de la imagen agrupando los píxeles que tienen unas características de niveles de gris similares. Algunas técnicas que usan esto son la umbralización, el crecimiento de regiones...

A diferencia de las redes de **detección de objetos** estas redes trabajan a nivel de píxel. Es decir, son capaces de indicar dónde se encuentra exactamente el objeto que se desea clasificar dibujando su contorno. Pudiendo obtener así mejores resultados a la hora de obtener el porcentaje de coral vivo/muerto de una manera más rigurosa.



(a) Detección de objetos (b) Segmentación de imagen

Figura 2.8: Diferencia entre detección de objetos y segmentación semántica[37]

Como se puede ver en la **Figura 2.8a** (detección de objetos) se marca un recuadro alrededor de cada objeto detectado, asignando el mismo color a los objetos que correspondan a la misma clase.

Sin embargo, en la **Figura 2.8b** (segmentación de imagen) se puede apreciar el contorno a nivel de píxel de cada objeto clasificado.

2.5.1. Tipos de segmentación de imagen[12]

La segmentación de imágenes es uno de los procesos más importantes de procesado de imágenes. Es una técnica usada para dividir o particionar una imagen en partes, llamadas segmentos. La segmentación es la técnica más utilizada para aplicaciones como la comprensión de imágenes o reconocimiento de objetos, porque para estos tipos de aplicaciones es ineficiente procesar toda la imagen. Con esta técnica divide a la imagen en varias partes en función de ciertas características de imagen, como el valor de intensidad de píxeles, el color, la textura, etc.

Los principales tipos de segmentación de imagen son los siguientes:

- **Clasificación de imagen:** Este es el bloque más elemental donde, dada una imagen, esperamos que el modelo nos proporcione una etiqueta con la clase del objeto principal de la imagen. En la clasificación de imágenes, se asume que hay un único objeto en la imagen y no varios como en algunos de los siguientes tipos de segmentación.
- **Clasificación + localización:** En este método se espera la respuesta por parte del modelo de la localización del objeto además de la clasificación de este. Es decir, buscamos conocer dónde se encuentra el objeto exactamente dentro de la imagen. Esta localización se implementa normalmente mediante un cuadro, incluso en este caso, se supone que solo hay un objeto por imagen.
  - **Detección de objetos:** La Detección de objetos[22] extiende la localización al siguiente nivel, es decir, ahora las imágenes pueden contar con más de un

objeto y más de una clase. La idea principal, al igual que en el punto anterior, es tanto clasificar por clases, como localizar todos los objetos de una imagen.

- **Segmentación semántica:** El objetivo de la segmentación semántica[6] es el etiquetado de una imagen a nivel de píxel; el modelo debe indicar a la clase que representa cada píxel de una imagen. Debido a que estamos prediciendo para cada píxel en la imagen, esta tarea se conoce comúnmente como predicción densa.
- **Segmentacion de instancias**: La segmentación de instancias está un paso adelante de la segmentación semántica. En este caso, de la clasificación a nivel de píxel, esperamos que el modelo nos clasifique cada objeto de una clase por separado.

2.6. Transfer learning

Transfer learning[19] es una técnica de ***Machine Learning*** que se basa en utilizar modelos ya entrenados en una tarea, para ser reentrenados en otra tarea relacionada. Esto nos permite una optimización más rápida de la red al aprovechar el conocimiento de la red en la tarea anterior.

Para este proyecto es interesante utilizar una red ya entrenada que tenga como parámetros de entrada imágenes, ya que para este caso, se le pasará como entrada imágenes del fondo marino.

Hay varios modelos de redes ya entrenadas que pueden ser incorporados al proyecto y esperan imágenes como datos de entrada. Por ejemplo:

- **Oxford VGG Model.**
- **Google Inception Model.**
- **Microsoft ResNet Model.**

En definitiva, **Transfer learning** es una manera de optimizar el tiempo obteniendo unos mejores resultados.

2.7. Google Colab

Para el entrenamiento de las redes buscando una mayor eficiencia y velocidad se ha decidido utilizar **Google Colab**[7], este es un servicio en la nube, que provee de una **Jupyter Notebook** (entorno de desarrollo muy extendido que permite sacar partido a la interactividad de Python) a la que accedemos a través de cualquier navegador web sin importar si usamos Windows, Linux o Mac. Además, sus principales ventajas son:

- Posibilidad de activar una GPU que aumenta la velocidad de cómputo de, por ejemplo, las múltiples multiplicaciones de matrices que se hagan a lo largo del entrenamiento.
- Tiene preinstaladas las bibliotecas comunes usadas en *datascience* y la posibilidad de instalar otras que necesitemos.
- Es gratuito.

Capítulo 3

Introducción a las plataformas y arquitecturas utilizadas durante el proyecto

3.1. Plataformas de aprendizaje automático

Actualmente, el aprendizaje automático está siendo tratado en diversos lenguajes de programación, sin embargo, es **Python** el que lidera el ranking de lenguajes más actualizados en el año 2020. En los últimos tres años ha sufrido un aumento de popularidad frente a lenguajes que hace poco tiempo estaban en la cima como puede ser R. Otros lenguajes que son utilizados actualmente, pero en mucha menor medida, son: Java, C/C++ o Matlab, que en los últimos años está potenciado diversas aplicaciones con sus actualizaciones anuales.

En este caso, se decidió utilizar Python como lenguaje debido a que si se echa un vistazo a la gráfica de la **Figura 3.1** del año 2018, de los 11 **frameworks** más utilizadas, 10 de ellos trabajan con Python. Los más populares son TensorFlow, Keras, PyTorch y Caffe. A lo largo del proyecto se utilizarán dos de estas, **Pytorch** y **Keras**.

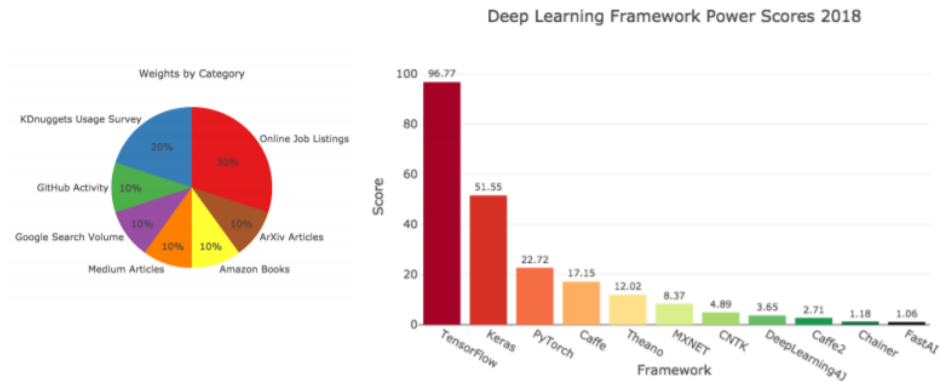


Figura 3.1: Comparativa de las principales plataformas de aprendizaje automático. *Imagen de* ***Towards Data Science***

Un estudio de uso, interés y popularidad, otorgó las puntuaciones de la **Figura 3.3** a las plataformas más utilizadas, basándose en las demandas de empleo, la encuesta de uso de KDnuggets, las publicaciones (Medium, Amazon Books, ArXiv) y la actividad en GitHub.

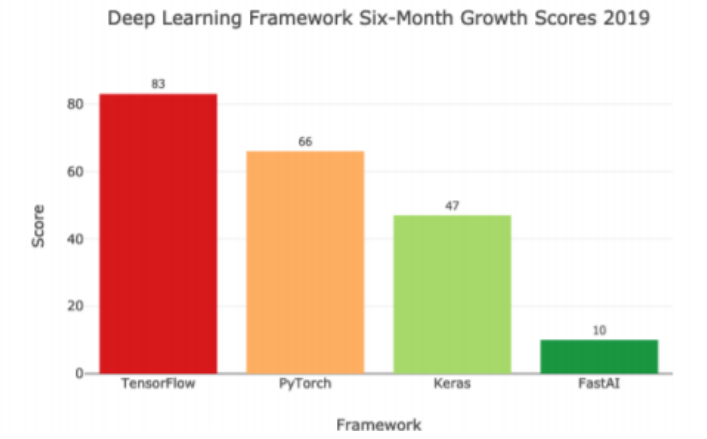


Figura 3.2: Plataformas con mejor puntuación por parte de los usuarios en 2019. *Imagen de* ***Pinterest***

Tensorflow, respaldado por Google, y Pytorch, con el respaldo de Facebook, son las dos plataformas independientes que mejor puntuación han obtenido, y ambas están disponibles para Python. Aunque Tensorflow es el ganador indiscutible, Pytorch está creciendo mucho en el último año.

3.2. Modelo de segmentación de imagen

3.2.1. U-NET

U-Net[16] consiste en varias capas de convolución, **Max Pooling**, **ReLU**, **concatenación** y **up sampling** que se reparten en tres secciones: contracción, cuello de botella y por último expansión. Esto le da una forma piramidal de izquierda a derecha y la inversa del cuello de botella hacia la expansión.

La sección denominada contracción está compuesta por 4 bloques, cada uno de estos bloques aplica dos 3x3 capas de convolución ReLU y después un 2x2 max pooling. El número de **feature maps** por tanto, se duplica en esa capa de pooling.

La sección de cuello de botella aplica dos 3x3 capas de convolución y 2x2 capas de convolución.

Mientras tanto, la sección de expansión consiste en varios bloques de expansión con cada bloque pasando dos capas Conv 3x3 y una capa de muestreo superior 2x2, que reduce a la mitad el número de canales de características. También incluye una concatenación con el mapa de características recortado el aumento correspondientemente de la ruta de contratación.

Finalmente, la capa 1x1 Conv se usa para hacer que el número de mapas de características sea igual al número de segmentos que se desean en la salida. U-net utiliza una función de pérdida para cada píxel de la imagen. Esto ayuda a identificar fácilmente las celdas individuales dentro del mapa de segmentación. Softmax se aplica a cada píxel seguido de una función de pérdida. Esto convierte el problema de segmentación en un problema de clasificación en el que debemos clasificar cada píxel en una de las clases.

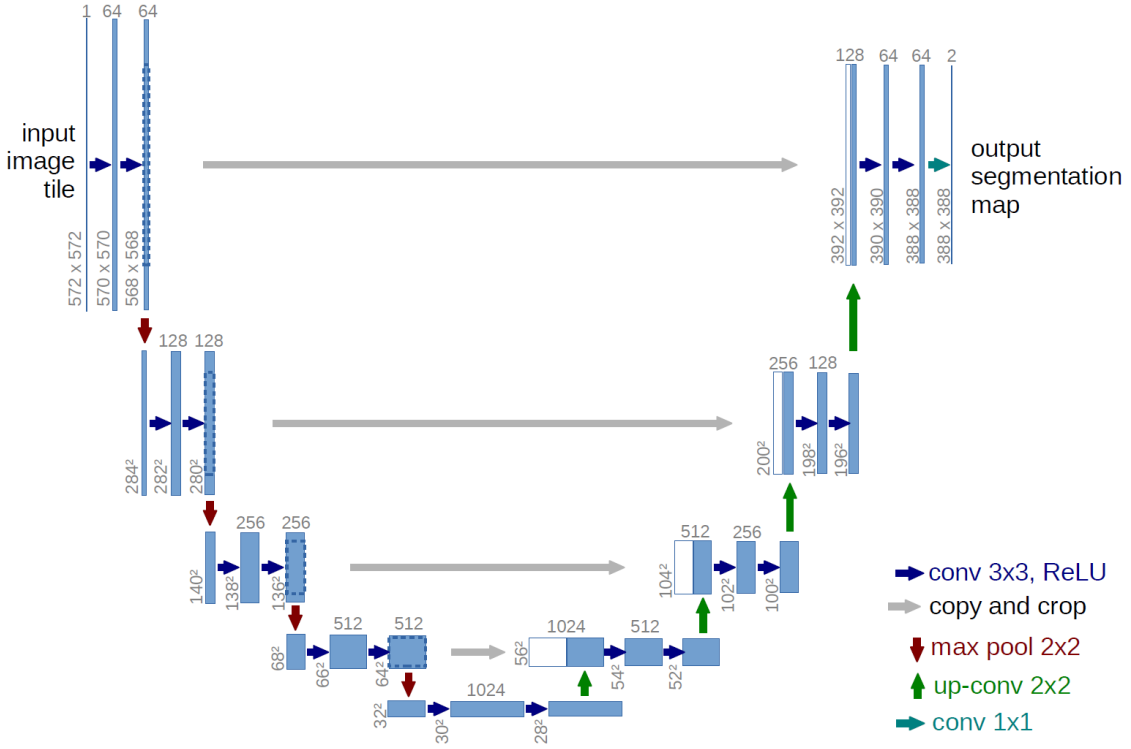


Figura 3.3: Arquitectura U-Net [19]

3.3. Modelo de segmentación de instancias

3.3.1. Detectron2

Detectron2[11] es un sistema de *software* de inteligencia artificial de **Facebook** que implementa algoritmos de detección de objetos y máscaras para segmentación semántica de última generación. Se basa en tecnología **MASK-RCNN** lo que quiere decir que además de la segmentación semántica a nivel de píxel incluye la detección del objeto.

MASK-RCNN[7] está basado en **PyTorch**, es un modelo que trabaja dos de los aspectos de la segmentación de imagen que se enumeraron en el **Punto** **2.5.1**, la detección de objetos (utilizando principalmente la red conocida como **Resnet** con sus diferentes opciones) y la segmentación de instancias. Es decir, completaremos tres aspectos mencionados en el **Punto** **2.5.1** ya que la red U-NET anteriormente mencionada se basa en la segmentación semántica.

La **Figura** **3.4** detalla la arquitectura del modelo MASK-rcnn.

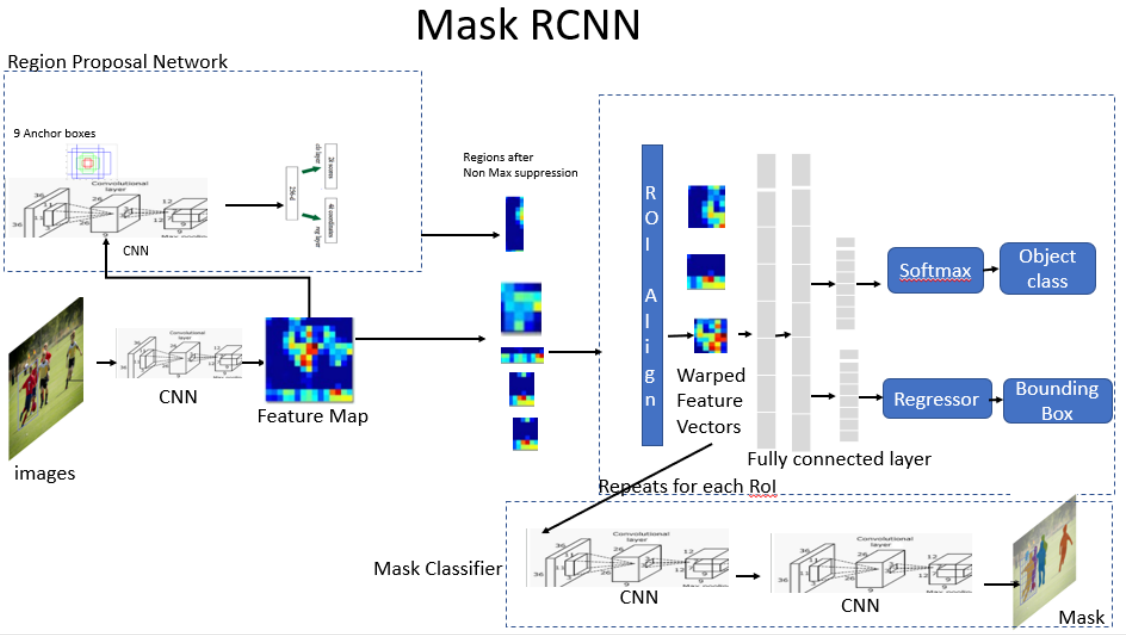


Figura 3.4: MASK-RCNN [6]

Pytorch

El modelo **Detectron2** basa su funcionamiento en la biblioteca **PyTorch**, una alternativa a las bibliotecas más utilizadas como Tensorflow o Keras[12].

PyTorch está sufriendo un gran crecimiento en los últimos años debido a sus diversas ventajas, su facilidad de uso y la capacidad nativa de ejecución basada en GPU o tarjeta gráfica, que acelera los procesos de entrenamiento de los modelos y lo convierte en un exponente en el sector.

Residual Network (ResNet)

En las redes neuronales tradicionales, más capas significan una mejor red, pero debido al problema del gradiente de fuga, los pesos de la primera capa no se actualizarán correctamente a través de la propagación inversa. Como el gradiente de error se propaga a las capas anteriores, la multiplicación repetida hace que el gradiente sea pequeño. Por lo tanto, con más capas en las redes, su rendimiento se satura y comienza a empeorar su desempeño rápidamente. Sin embargo, Res-Net[8] resuelve este problema utilizando la matriz de identidad. Cuando la retropropagación se realiza mediante la función de identidad, el gradiente se multiplicará solo por 1. Esto preserva la entrada y evita cualquier pérdida de información. Significando normalmente un mejor desempeño de los algoritmos cuando estos trabajan con muchas capas, sacrificando la velocidad de predicción.

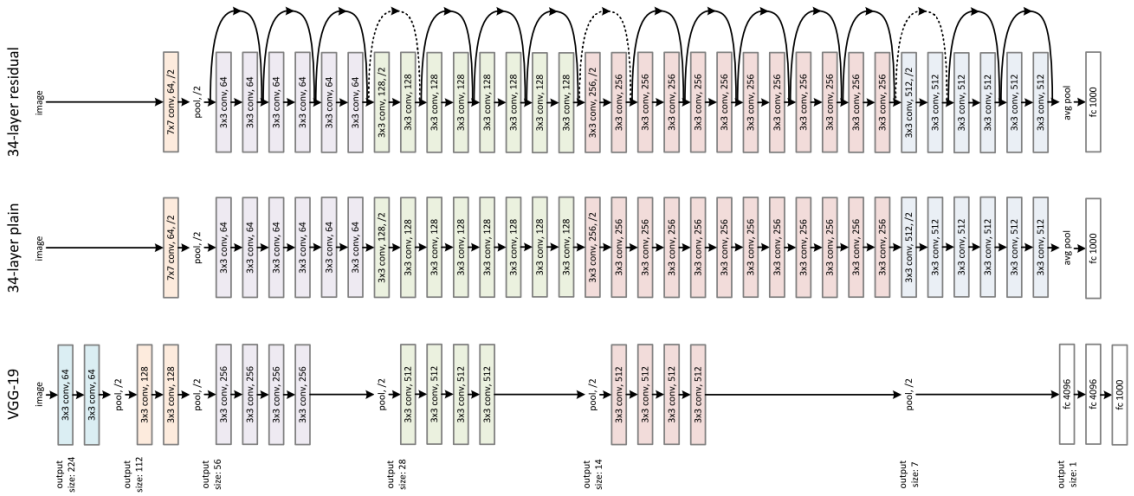


Figura 3.5: Arquitectura ResNet [10]

Los componentes de una red incluyen filtros 3x3, capas de muestreo descendente CNN con stride 2, capa de agrupación promedio global y una capa totalmente conectada de 1000 vías con softmax al final.

ResNet utiliza una conexión de omisión en la que también se agrega una entrada original a la salida del bloque de convolución. Esto ayuda a resolver el problema de la desaparición del gradiente al permitir una ruta alternativa para que el gradiente fluya. Además, utilizan la función de identidad que ayuda a que la capa superior funcione tan bien como una capa inferior, y no peor.

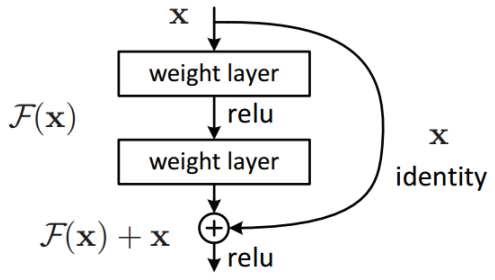


Figura 3.6: Bloque residual.

En las redes neuronales tradicionales, cada capa se alimenta a la siguiente capa. Pero en una red con bloques residuales, cada capa se alimenta a la siguiente capa y directamente a las capas a unos saltos de distancia.

3.4. Arquitectura del modelo utilizado para detección de objetos

3.4.1. YOLO[34]

El modelo **You Only Look Once** (YOLO), es un sistema de código abierto implementado en **Python** utilizado para la detección de objetos en tiempo real permitiendo así su uso en por ejemplo, cámaras de tráfico y demás streamings.

Este modelo hace uso de una única red neuronal convolucional para detectar objetos en imágenes.

La red neuronal divide la imagen en varias regiones, prediciendo cuadros de identificación y probabilidades por cada región a las clases especificadas; las cajas son ponderadas a partir de las probabilidades predichas, siendo la de mayor probabilidad la caja que finalmente visualiza el usuario. El algoritmo aprende a generalizar información de los objetos, consiguiendo así disminuir los errores de detección en objetos diferentes al conjunto de datos de entrenamiento.

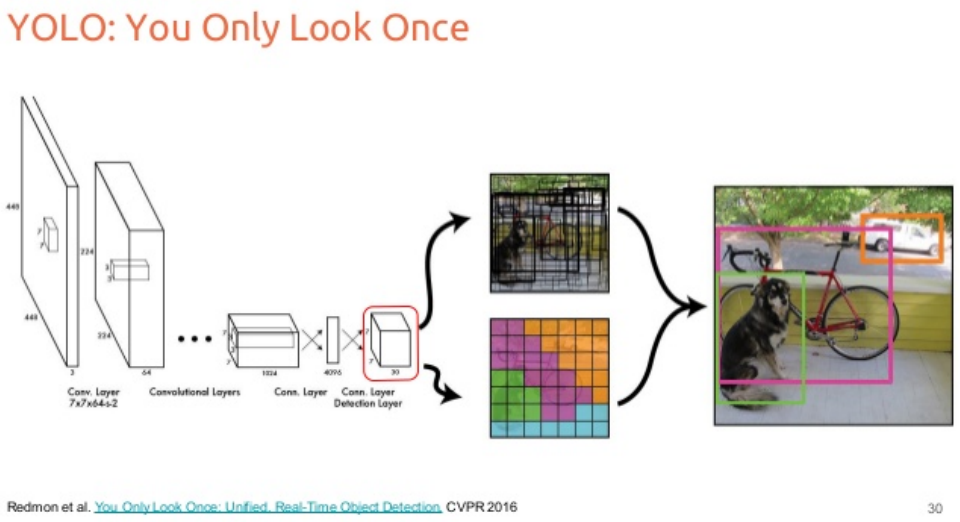


Figura 3.7: YOLO.

Las capas convolucionales iniciales de la red se encargan de la extracción de características comunes entre las imágenes del set de entrenamiento, mientras que las capas de conexión completa predicen la probabilidad de salida y las coordenadas del objeto en la imagen.

La red tiene 24 capas convolucionales seguidas por 2 capas de conexión completa; esta hace uso de capas de reducción de 1x1 seguidas de capas convolucionales de 3x3.

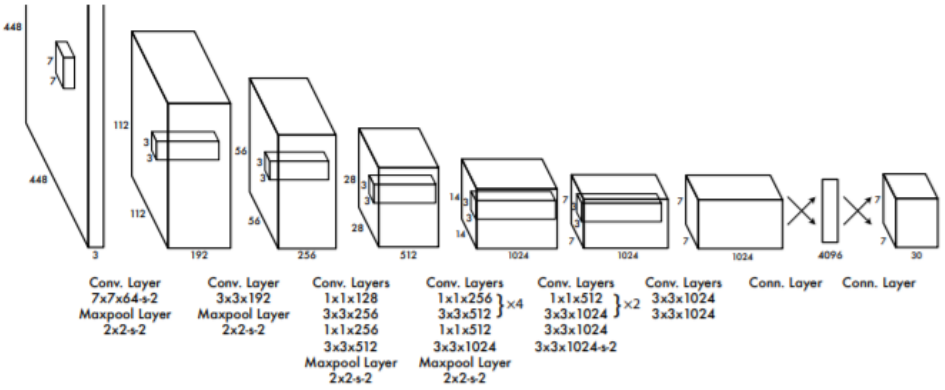


Figura 3.8: Arquitectura YOLO [34]

YOLOv4[33]

Para el proyecto se utilizó finalmente la versión 4 del modelo en el que la arquitectura de YOLO se mantiene. Sin embargo, se introducen varias mejoras para el proceso de entrenamiento. Por ejemplo, aumento de datos, mejores resultados con desbalanceo de clases, mejora de la función de costes, etc. Gracias a esto, se obtiene una mejor precisión que en versiones anteriores superando a las **fast R-CNN** tanto en velocidad como en precisión.

Capítulo 4

Desarrollo Práctico

4.1. Procedimiento para modelo Detectron2

4.1.1. Selección de imágenes y etiquetado

Una de las partes más importantes del proyecto es la selección y el etiquetado de las imágenes que formarán parte del futuro ***dataset*** de trabajo. En este caso, el ***dataset*** fue obtenido a través de un vídeo con resolución **1080x1920** prestado por el oceanográfico. Este fue separado en los diferentes fotogramas que componen el vídeo gracias a un script de **Matlab** (**Anexo A.4**) y se fueron seleccionando dichos fotogramas con varios saltos en el tiempo, obteniendo así un set de datos lo más amplio posible, buscando la mayor variedad de datos posible.

4.1.2. Etiquetado

En este caso, para el etiquetado se utilizó la herramienta ***Labelme***[14]. Se trata de una herramienta de etiquetado que trabaja sobre polígonos, se basa en Python y es muy útil y popular debido a su simplicidad de uso e interfaz.

En la **Figura 4.1** se aprecian algunos ejemplos del etiquetado con dicha herramienta. Los corales vivos son etiquetados en color verde, mientras que los muertos en rojo, dibujando siempre el contorno de los mismos con polígonos con la mejor fidelidad a la realidad posible.

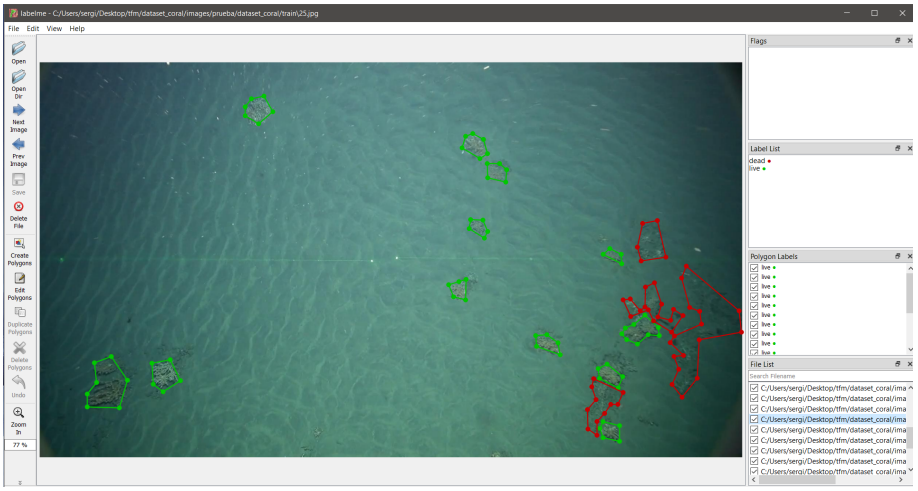


Figura 4.1: Etiquetado de uno de los fotogramas del vídeo con la herramienta Labelme.

El formato por defecto que se obtiene del etiquetado con Labelme es un archivo .json, por cada imagen etiquetada en él, se detallan los puntos de cada polígono, la clase que ha sido etiquetada y más información relevante. Sin embargo, en la gran mayoría de herramientas para segmentación semántica se requiere un formato propietario como el de **COCO dataset** que es de igual forma .json y consta de un archivo en el que se representa la principal información necesaria para el entrenamiento.

Para conseguir este formato se hizo uso del script **labelme2coco.py** del **Anexo A.1** al que se le pasa como argumento la carpeta en la que se encuentran tanto las imágenes como los .json de cada archivo, obteniéndose así un único .json con la siguiente información:


```
"images": [
    {
        "height": 1080,
        "width": 1920,
        "id": 0,
        "file_name": "1.jpg"
    },
    {
        "height": 1080,
        "width": 1920,
        "id": 1,
        "file_name": "10.jpg"
    },
    {
        "height": 1080,
        "width": 1920,
        "id": 2,
        "file_name": "11.jpg"
    },
```

(a) Nombre, id y tamaño.

```
{
    "segmentation": [
        [
            395.8780487804878,
            69.3170731707317,
            399.5365853658536,
            142.48780487804876,
            442.2195121951219,
            164.4390243902439,
            476.36585365853654,
            198.58536585365852,
            532.4634146341463,
            171.7560975609756,
            521.4878048780488,
            126.63414634146341,
            497.0975609756097,
            112.0,
            459.29268292682923,
            97.36585365853657,
            483.68292682926824,
            83.95121951219511,
            454.4146341463414,
            68.09756097560975,
            449.5365853658536,
            61.999999999999986,
            438.56097560975604,
            48.58536585365853,
            422.70731707731707,
            66.8780487804878
        ]
    ],
```

(b) Puntos para segmentación

```
    "iscrowd": 0,
    "area": 11337.745389649004,
    "image_id": 3,
    "bbox": [
        395.0,
        48.0,
        137.0,
        150.0
    ],
    "category_id": 1,
    "id": 52
},
```

(c) Cajas para detección

Figura 4.2: Formato archivo .Json

Como se puede ver en las imágenes, en la **Figura 4.2b** se relatan todas las imágenes del *dataset* etiquetado, se les da un número de identificación a cada una de ellas y se menciona el tamaño de la misma.

Este número de identificación es interesante ya que más adelante se usará para identificar los puntos de los polígonos etiquetados para la segmentación (**Figura 4.2b**) y además los puntos de las bbox (**Figura 4.2c**).

4.1.3. Primer dataset

Debido a la gran cantidad de elementos a etiquetar se optó por crear un primer *dataset* que contenía únicamente 35 fotogramas. Esto se podría considerar un *dataset* muy pobre, sin embargo, sumando los objetos etiquetados, tanto coral muerto, como coral vivo, el número de objetos que se encontraba en esos 35 fotogramas era de 650, con corales tanto vivos (518) como muertos (132), de diferentes formas, tamaños, y escalas de color.

category	#instances	category	#instances
dead	132	live	518
total	650		

Figura 4.3: Número de objetos de cada clase y total de objetos para el entrenamiento.

4.1.4. Detectron2

Instalar dependencias

Lo primero que hay que hacer para trabajar con **Detectron2** es instalar las dependencias que se requerirán a lo largo del proyecto. En este caso, se trabajó sobre Google Colab trabajando sobre las GPU, que esta herramienta nos regala de forma gratuita.

```
1 # install dependencies: (use cu100 because colab is on CUDA 10.0)
2 !pip install -U torch==1.4+cu100 torchvision==0.5+cu100 -f https://download.pytorch
      .org/whl/torch_stable.html
3 !pip install cython pyyaml==5.1
4 !pip install -U 'git+https://github.com/cocodataset/cocoapi.git#subdirectory=
      PythonAPI'
5 import torch, torchvision
6 torch.__version__
7 !gcc --version
```

Código 4.1: `Instalación de depencencias`

Instalar detectron2

El siguiente paso es instalar el *software* de detectron2, en Google Colab:

```
# install detectron2:
!pip install detectron2 -f https://dl.fbaipublicfiles.com/detectron2/wheels/cu100/
    index.html
```

Código 4.2: `Instalación Detectron2.`

4.1.5. Registro del dataset

Este software requiere un *dataset* de formato COCO, mencionado anteriormente; pero como curiosidad respecto a otras herramientas de *software*, es necesario registrar nuestro *dataset* en su software si el *dataset* que se va a utilizar no es el de COCO, que actualmente cuenta con 80 clases diferentes en las que se encuentran, por ejemplo, personas, gatos, perros, caballos, etc.

En nuestro caso, al tratarse de dos clases no demasiado comunes, debemos trabajar con lo que se llama un **custom dataset** por lo que debe ser registrado.

```
from detectron2.data.datasets import register_coco_instances

for d in ["train", "val"]:
    register_coco_instances(f"dataset_coral_segmentation_{d}", {}, f"
        dataset_coral_segmentation/{d}.json", f"dataset_coral_segmentation/{d}")
```

Código 4.3: `Registro del dataset.`

```
import random
from detectron2.data import DatasetCatalog, MetadataCatalog

dataset_dicts = DatasetCatalog.get("dataset_coral_segmentation_val")
coral_metadata = MetadataCatalog.get("dataset_coral_segmentation_train")

for d in random.sample(dataset_dicts, 3):
    img = cv2.imread(d["file_name"])
    v = Visualizer(img[:, :, ::-1], metadata=coral_metadata, scale=1)
    v = v.draw_dataset_dict(d)
    plt.figure(figsize = (14, 10))
    plt.imshow(cv2.cvtColor(v.get_image()[:, :, ::-1], cv2.COLOR_BGR2RGB))
    plt.show()
```

Código 4.4: `Visualización del dataset.`

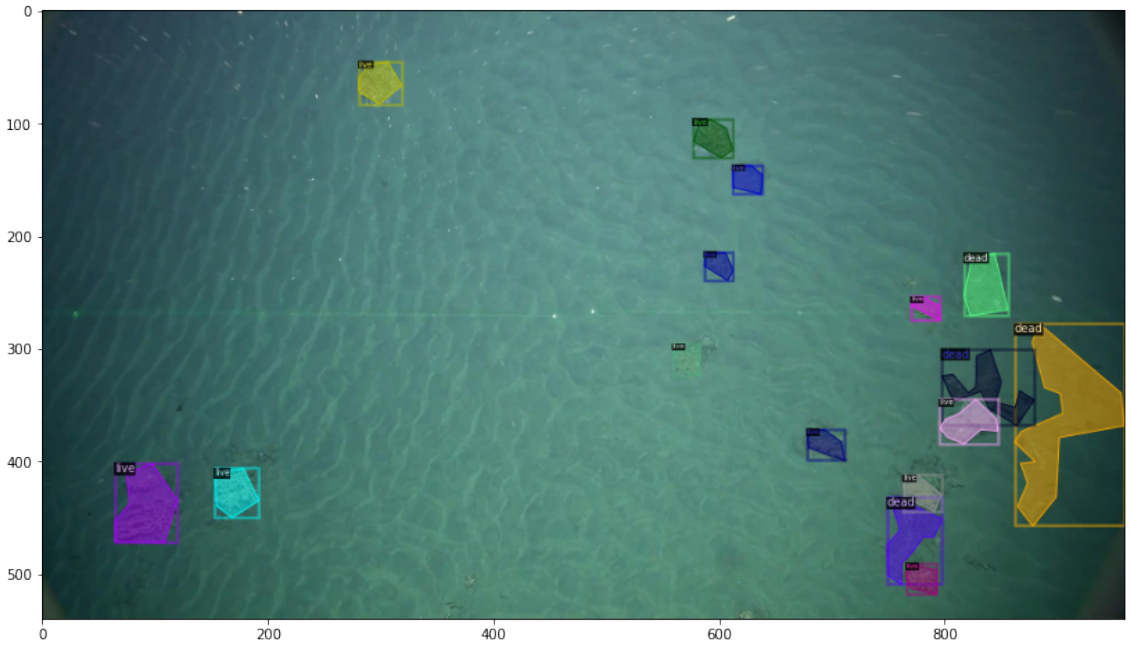


Figura 4.4: Registro del *dataset* para el posterior entrenamiento del modelo.

En la **Figura 4.4** se puede ver un ejemplo de una de las imágenes del *dataset* ya registrado en **Detectron2**.

4.1.6. Entrenamiento del primer modelo: maskrcnn + resnet50

El siguiente paso es entrenar el modelo, en este caso se utilza una **faster RCNN**

```
from detectron2.engine import DefaultTrainer
from detectron2.config import get_cfg
import os

cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_file("COCO-InstanceSegmentation/
    mask_rcnn_R_50_FPN_3x.yaml"))
cfg.DATASETS.TRAIN = ("dataset_coral_segmentation_train",)
cfg.DATASETS.TEST = ()
cfg.DATALOADER.NUM_WORKERS = 2
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-InstanceSegmentation/
    mask_rcnn_R_50_FPN_3x.yaml")
cfg.SOLVER.IMS_PER_BATCH = 2
cfg.SOLVER.BASE_LR = 0.00025
cfg.SOLVER.MAX_ITER = 1000
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 2

os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
trainer = DefaultTrainer(cfg)
trainer.resume_or_load(resume=False)
```

```
19 trainer.train()
```

Código 4.5: `Entrenamiento del modelo.`

En la anterior imagen, se puede ver cómo se detalla la configuración del algoritmo, en este caso, vamos a utilizar el fichero de configuración del modelo ***mask_rcnn_R_50_FPN_3x .yaml***. Este modelo es una mezcla entre los dos tipos de ***Machine Learning*** mencionados al comienzo de este documento. En primer lugar usamos mask_rcnn, el cual trabaja con las máscaras ya mencionadas, consiguiendo así el etiquetado a nivel de píxel. En segundo lugar, tenemos el uso de la red **resnet50**, muy conocida y utilizada en detección de objetos debido a su precisión y velocidad en la detección. El *dataset* utilizado es el mencionado anteriormente, un *dataset* etiquetado manualmente por la herramienta **Labelme** que consta de 35 imágenes etiquetadas, tanto a nivel de píxel como de localización en ***Bounding boxes***.

Además, se utilizan los pesos característicos del modelo como base de entrenamiento de nuestro modelo, ahorrándonos una gran cantidad de tiempo del entrenamiento. Entrenar un modelo desde 0 requeriría entrenar cada una de las capas del modelo (en este caso 50 al tratarse de una **resnet 50)** y no conllevaría una mejora sustancial de los resultados finales, ya que muchos modelos tienen capas iniciales comunes que son siempre iguales en la detección de cualquier tipo de objeto.

También se le indica el ***learning rate*** o tasa de aprendizaje del modelo. En este caso se queda fija en 0.00025, el número de imágenes que se le pasará a la GPU en cada batch es 2, el número máximo de iteraciones es 1000, este es el valor por defecto que se recomienda en el artículo del autor de detectron2 para el número de clases que se va a entrenar en este caso, sin embargo, puede ocurrir que se obtenga mejores resultados con más iteraciones, pero siempre según el artículo con 1000 iteraciones ya tendríamos un modelo funcional y con buenas métricas. Sin embargo, el coral que vamos a detectar no es un objeto tan común para la red como podría ser un perro o una persona, por lo tanto, es posible que se requiera un mayor número de iteraciones.

Una vez conocidos todos los parámetros de la configuración, los resultados del entrenamiento tras 1000 iteraciones, se pueden ver en la **Figura 4.5**:

```
 eta: 0:00:27  iter: 939  total_loss: 1.163  loss_cls: 0.301  loss_box_reg: 0.466  loss_mask: 0.295  loss_rpn_cls: 0.019  loss_rpn_loc: 0.097
 eta: 0:00:18  iter: 959  total_loss: 1.186  loss_cls: 0.300  loss_box_reg: 0.466  loss_mask: 0.302  loss_rpn_cls: 0.020  loss_rpn_loc: 0.095
 eta: 0:00:09  iter: 979  total_loss: 1.175  loss_cls: 0.291  loss_box_reg: 0.463  loss_mask: 0.292  loss_rpn_cls: 0.022  loss_rpn_loc: 0.104
 eta: 0:00:00  iter: 999  total_loss: 1.160  loss_cls: 0.292  loss_box_reg: 0.442  loss_mask: 0.294  loss_rpn_cls: 0.021  loss_rpn_loc: 0.104
Overall training speed: 997 iterations in 0:07:23 (0.4446 s / it)
Total training time: 0:07:27 (0:00:04 on hooks)
```

Figura 4.5: Resultado del entrenamiento del modelo con resnet50 para 1000 iteraciones.

Como se puede ver, las pérdidas totales se quedan en 1.16, sin embargo, en las últimas iteraciones, estas pérdidas siguen descendiendo, por lo tanto, nuestro modelo sigue aprendiendo, por lo que podría ser una posible mejora en el rendimiento del modelo si lo dejásemos entrenando varias iteraciones más, al menos hasta que las pérdidas totales

bajen de 1, un número de pérdidas "mágico" en numerosos artículos sobre detección de objetos. Pero antes de empezar a hacer cambios, trataremos de obtener las métricas con estos nuevos pesos entrenados para ver si merece o no la pena un aumento en el tiempo de entrenamiento.

4.1.7. Inferencia del primer modelo entrenado tras 1000 iteraciones

En este punto, teniendo ya entrenado un modelo con unos pesos resultantes se hace la inferencia con estos mismos:

```
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth")
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5
cfg.DATASETS.TEST = ("dataset_coral_segmentation_test", )
predictor = DefaultPredictor(cfg)
```

Código 4.6: `Configuración de la inferencia.`

El código de la parte superior corresponde a la configuración en la que se señala, los pesos que serán utilizados para la inferencia, en este caso los que tienen de nombre **model_final.pth** (los pesos obtenidos del entrenamiento del punto anterior), el umbral de decisión o ***threshold*** para la inferencia, es decir, a partir de qué punto se tomará como cierto un objeto o no. En este caso a partir del 50 % de seguridad se dará por válido. Además hay que detallarle el *dataset* sobre el que tiene que hacer la inferencia, en este caso el set de validación que preparamos manualmente para poder apreciar así las diferencias que se puedan encontrar entre el etiquetado manual y el etiquetado del modelo.

```
from detectron2.utils.visualizer import ColorMode
dataset_dicts = DatasetCatalog.get("dataset_coral_segmentation_train")
for d in random.sample(dataset_dicts, 3):
    im = cv2.imread(d["file_name"])
    outputs = predictor(im)
    v = Visualizer(im[:, :, ::-1],
                   metadata=microcontroller_metadata,
                   scale=0.8,
                   instance_mode=ColorMode.IMAGE_BW   # remove the colors of
                       unsegmented pixels
    )
    v = v.draw_instance_predictions(outputs["instances"].to("cpu"))
    plt.figure(figsize = (14, 10))
    plt.imshow(cv2.cvtColor(v.get_image()[:, :, ::-1], cv2.COLOR_BGR2RGB))
    plt.show()
```

Código 4.7: `Inferencia.`

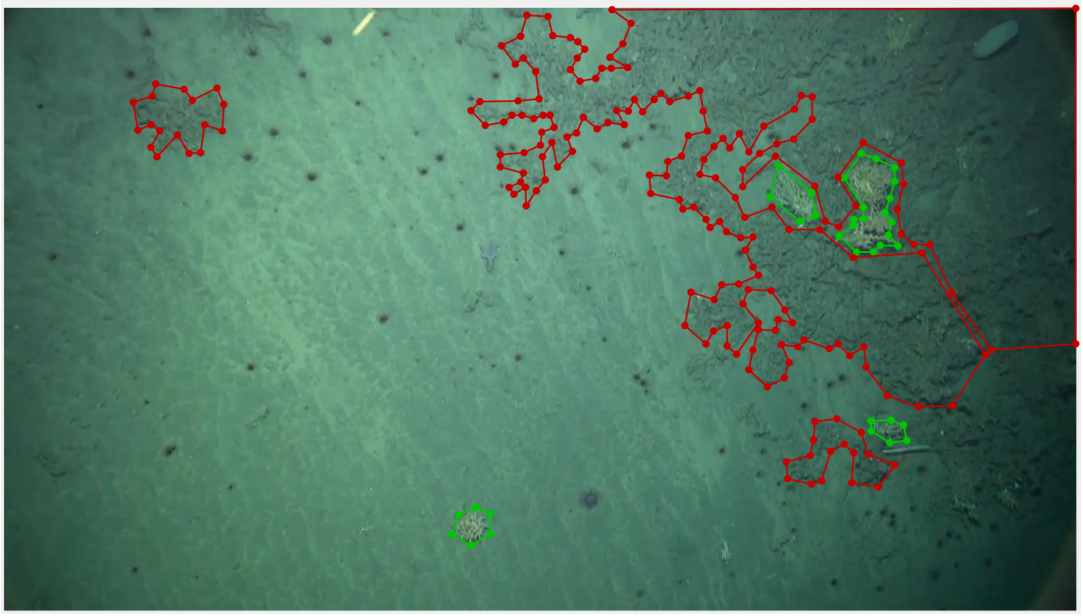


Figura 4.6: Etiquetado de uno de los fotogramas del vídeo con la herramienta Labelme.

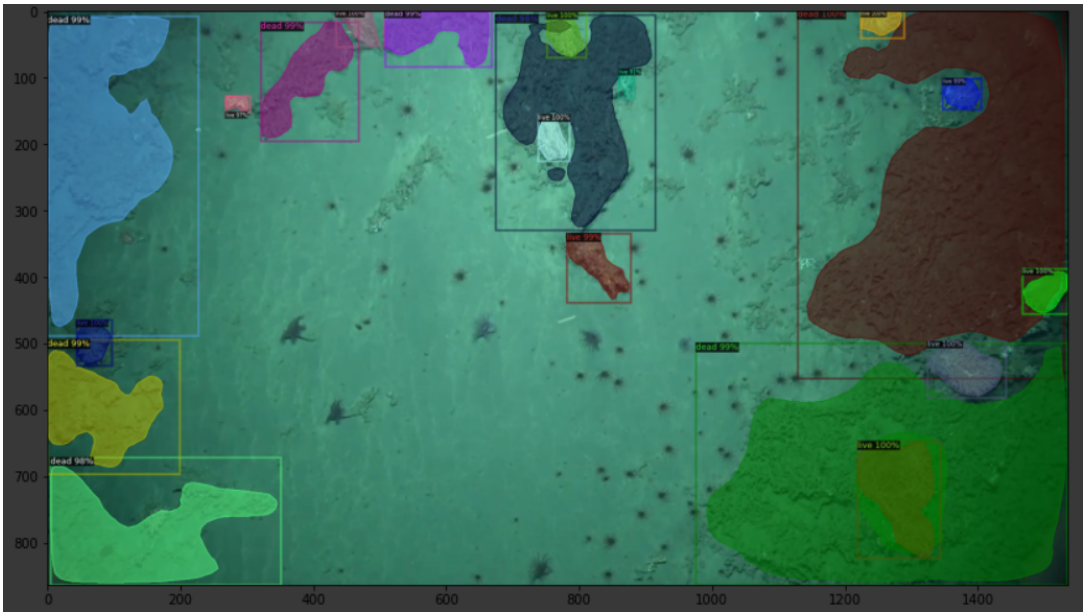


Figura 4.7: Inferencia del modelo entrenado sobre 1000 iteraciones.

Otro ejemplo de las diferencias entre la inferencia del modelo y el etiquetado original en otro fotograma diferente serían las dos siguientes figuras:

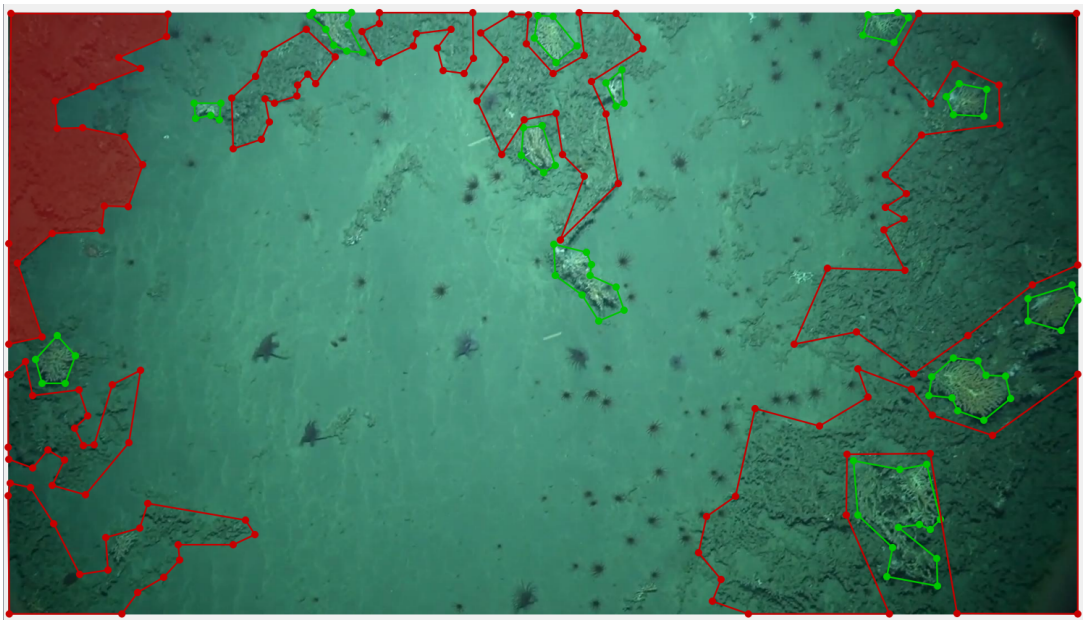


Figura 4.8: Etiquetado de uno de los fotogramas del vídeo con la herramienta Labelme.

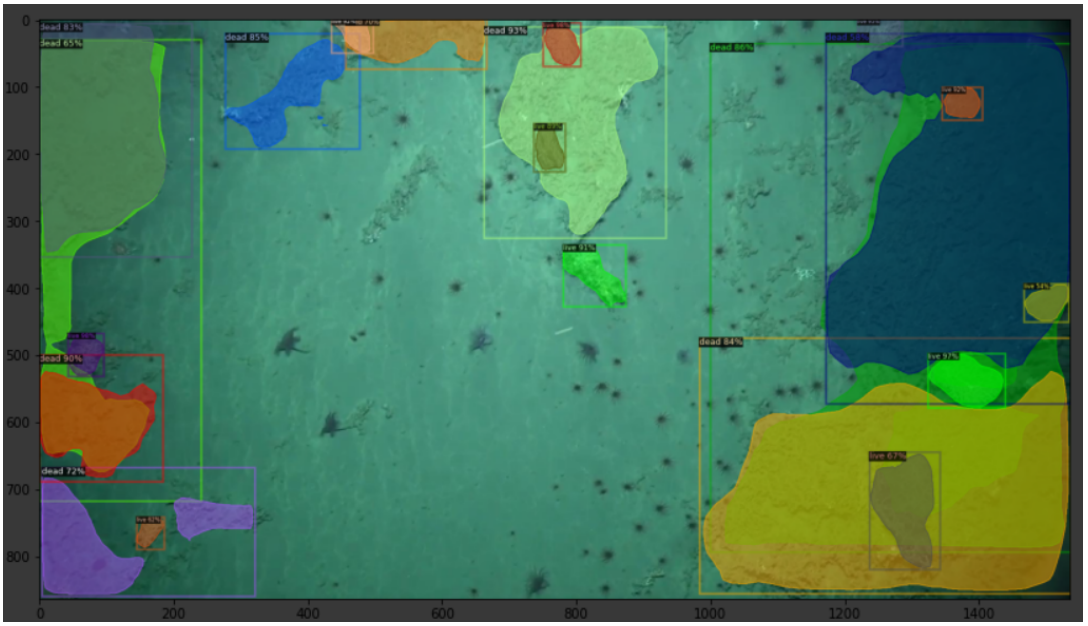


Figura 4.9: Inferencia del modelo entrenado sobre 1000 iteraciones.

Como se puede ver en la imagen superior, se trata de la misma imagen, sin embargo, la primera de ellas es el etiquetado manual, estando marcadas en rojo las zonas de coral muerto, mientras que en verde se encuentran las zonas de coral vivo.

En la segunda imagen, nos encontramos la inferencia del modelo entrenado, que como se puede apreciar detecta perfectamente las zonas de coral vivo pero tiene ligeros problemas a la hora de dibujar a nivel de píxel el coral muerto.

Introducción teórica a las métricas

La principal métrica que se va a observar en este trabajo para comparar entre los distintos modelos entrenados es la precisión. Por este motivo, se va a utilizar para el cálculo de las métricas el método COCO, este se basa en distintos aspectos de precisión con distintos umbrales de decisión.

- **Precisión y Recall** La precisión y el *Recall*[5] son dos de los parámetros más importantes en cuanto a métricas[9] de evaluación se refiere. Estos se traducen de la siguiente forma:

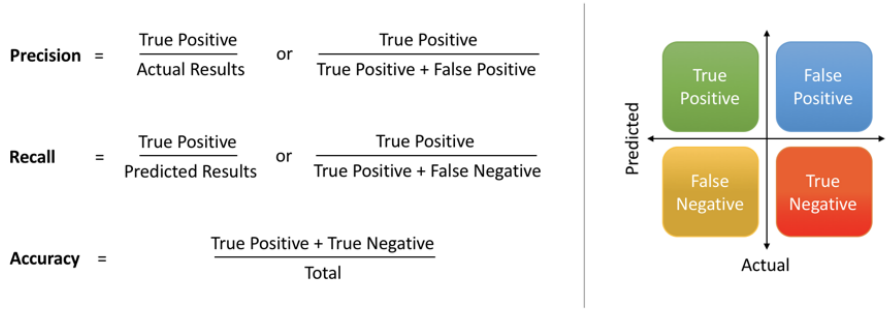


Figura 4.10: Definiciones de las métricas: *Precision* y *Recall*[11]

- **AP** Es la precisión media que se obtiene al inferir en el set al que se quiere hacer la métrica con tres umbrales distintos, 0.05, 0.5 y 0.95.
- **AP50** Es la precisión media que se obtiene al inferir en el set al que se quiere hacer la métrica con un umbral de decisión de 0.5.
- **AP75** Es la precisión media que se obtiene al inferir en el set al que se quiere hacer la métrica con un umbral de decisión de 0.75.
- **APs:** También conocido como AP_{small} se trata de la precisión media que se obtiene al inferir en el set al que se quiere hacer la métrica únicamente en los objetos pequeños.
- **APm:** También conocido como AP_{medium} se trata de la precisión media que se obtiene al inferir en el set al que se quiere hacer la métrica únicamente en los objetos medianos.
- **APl:** También conocido como AP_{large} se trata de la precisión media que se obtiene al inferir en el set al que se quiere hacer la métrica únicamente en los objetos grandes.

Métricas tras 1000 iteraciones

Una vez vista la inferencia sobre una única imagen, y que su funcionamiento se podría decir que es óptimo a simple vista, ahora vamos a pasar a conocer las métricas obtenidas sobre el set de validación para ver cómo de óptimo es realmente.

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.500
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.852
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.524
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.510
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.576
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.513
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.077
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.417
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.608
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.511
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.621
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.634
[06/06 22:36:47 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 50.037 | 85.185 | 52.352 | 50.990 | 57.615 | 51.343 |
[06/06 22:36:47 d2.evaluation.coco_evaluation]: Per-category bbox AP:
| category   | AP     | category   | AP     |
|:-----------|:-------|:-----------|:-------|
| dead       | 43.809 | live       | 56.265 |
```

Figura 4.11: Métricas para las bbox tras entrenar 1000 iteraciones.

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.374
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.792
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.296
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.426
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.366
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.394
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.053
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.294
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.451
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.456
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.476
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.457
[06/06 22:36:48 d2.evaluation.coco_evaluation]: Evaluation results for segm:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 37.354 | 79.198 | 29.552 | 42.598 | 36.602 | 39.389 |
[06/06 22:36:48 d2.evaluation.coco_evaluation]: Per-category segm AP:
| category   | AP     | category   | AP     |
|:-----------|:-------|:-----------|:-------|
| dead       | 23.152 | live       | 51.557 |
```

Figura 4.12: Métricas para la segmentación tras entrenar 1000 iteraciones.

4.1.8. Tratando de mejorar las métricas del primer modelo

Como se comentó varios párrafos atrás, 1000 iteraciones podrían ser pocas para un objeto tan "extraño" para la red como es el coral. Por lo tanto, con la idea de mejorar las prestaciones de las métricas, se optó por aumentar el número de iteraciones de entrenamiento, siendo el nuevo número máximo de 2000 y obteniéndose los siguientes nuevos resultados:

```
 eta: 0:02:00  iter: 1939  total_loss: 0.731  loss_cls: 0.148  loss_box_reg: 0.257  loss_mask: 0.238  loss_rpn_cls: 0.003
 eta: 0:01:21  iter: 1959  total_loss: 0.738  loss_cls: 0.157  loss_box_reg: 0.271  loss_mask: 0.242  loss_rpn_cls: 0.005
 eta: 0:00:41  iter: 1979  total_loss: 0.743  loss_cls: 0.148  loss_box_reg: 0.273  loss_mask: 0.241  loss_rpn_cls: 0.005
 eta: 0:00:01  iter: 1999  total_loss: 0.752  loss_cls: 0.141  loss_box_reg: 0.282  loss_mask: 0.243  loss_rpn_cls: 0.006
Overall training speed: 1997 iterations in 1:05:19 (1.9625 s / it)
Total training time: 1:05:26 (0:00:07 on hooks)
```

Figura 4.13: Resultados del entrenamiento tras 2000 iteraciones con resnet50.

Como se puede apreciar, las pérdidas totales esta vez si que bajan del número "mágico" 1, por lo tanto, el modelo ha aprendido más y teóricamente la inferencia del modelo será mejor. En el siguiente apartado se verá si es cierta esta afirmación.

4.1.9. Inferencia del primer modelo entrenado tras 2000 iteraciones

En este punto, con el modelo entrenado durante 2000 iteraciones, se lleva a cabo la inferencia con los pesos resultantes.

```
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth")
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5
cfg.DATASETS.TEST = ("dataset_coral_segmentation_test", )
predictor = DefaultPredictor(cfg)
```

Código 4.8: `Configuración de la inferencia.`

El código de la parte superior representa la configuración en la que se indica, por este orden, los pesos que serán utilizados para la inferencia, en este caso los que tienen de nombre **model_final.pth** (los pesos obtenidos del entrenamiento del punto anterior), el umbral de decisión o ***threshold*** para la inferencia, es decir, a partir de qué punto se tomará como cierto un objeto o no. En este caso a partir del 50 % de seguridad se dará por válido. Además se selecciona el *dataset* sobre el que se lleva a cabo la inferencia, en este caso el set de validación que preparamos manualmente. Con esto, se busca apreciar las diferencias que se puedan encontrar entre el etiquetado manual y la inferencia del modelo.

```
from detectron2.utils.visualizer import ColorMode
dataset_dicts = DatasetCatalog.get("dataset_coral_segmentation_train")
for d in random.sample(dataset_dicts, 3):
    im = cv2.imread(d["file_name"])
    outputs = predictor(im)
    v = Visualizer(im[:, :, ::-1],
                   metadata=microcontroller_metadata,
                   scale=0.8,
                   instance_mode=ColorMode.IMAGE_BW   # remove the colors of
                       unsegmented pixels
    )
    v = v.draw_instance_predictions(outputs["instances"].to("cpu"))
    plt.figure(figsize = (14, 10))
    plt.imshow(cv2.cvtColor(v.get_image()[:, :, ::-1], cv2.COLOR_BGR2RGB))
    plt.show()
```

Código 4.9: `Inferencia.`

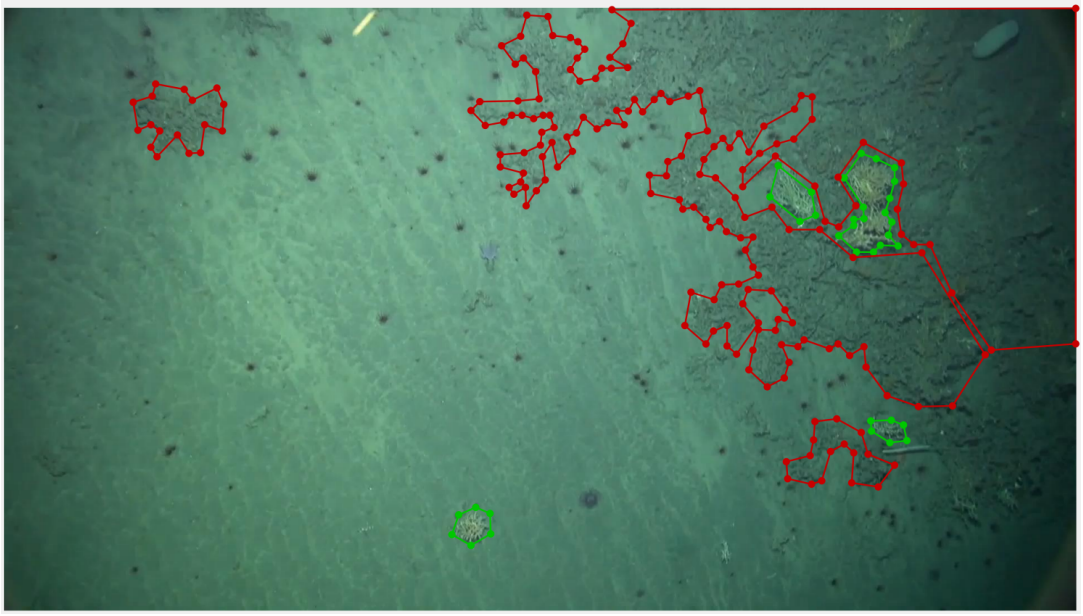


Figura 4.14: Etiquetado de uno de los fotogramas del vídeo con la herramienta Labelme.

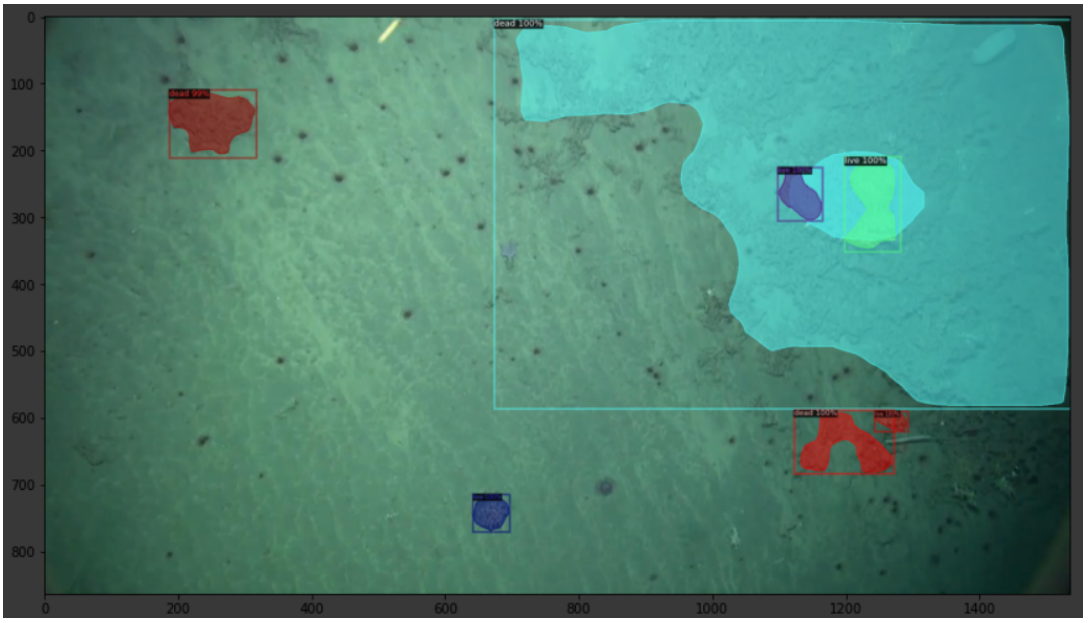


Figura 4.15: Inferencia del modelo resnet50 con 2000 iteraciones de entrenamiento.

Otro ejemplo de la inferencia del modelo en otro fotograma es el siguiente:

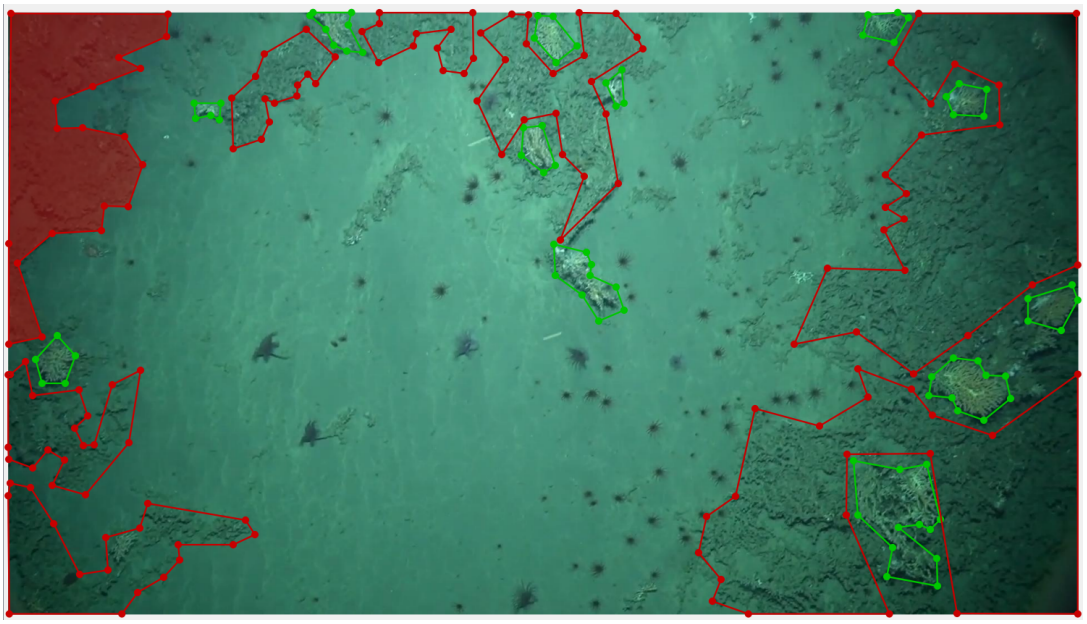


Figura 4.16: Etiquetado de uno de los fotogramas del vídeo con la herramienta Labelme.

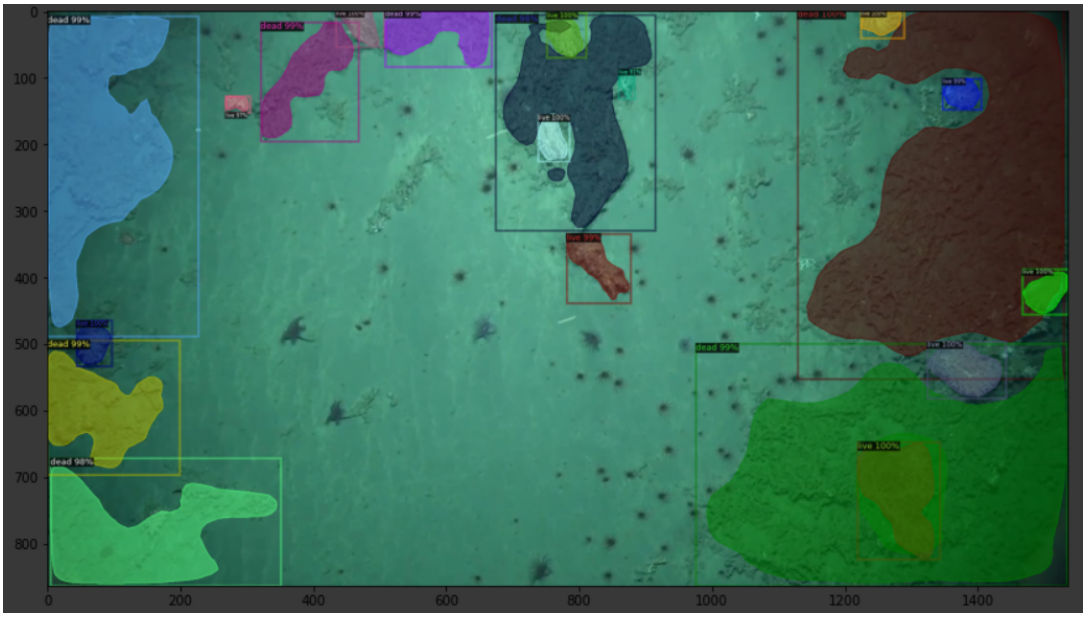


Figura 4.17: Inferencia del modelo resnet50 con 2000 iteraciones de entrenamiento.

Como se puede ver en las **Figuras** 4.16 y 4.17, se tratan de la misma imagen, sin embargo, la primera de ellas es el etiquetado manual, estando marcadas en rojo las zonas de coral muerto, mientras que en verde se enmarcan las zonas de coral vivo.

En la segunda imagen, nos encontramos la inferencia del modelo entrenado. Como se puede apreciar detecta perfectamente las zonas de coral vivo pero tiene ligeros problemas a la hora de dibujar a nivel de píxel el coral muerto.

Comentando los resultados de las métricas

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.766
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.999
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.935
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.680
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.781
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.797
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.098
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.595
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.807
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.700
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.804
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.846
[06/07 13:01:32 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 76.638 | 99.874 | 93.464 | 67.969 | 78.053 | 79.748 |
[06/07 13:01:32 d2.evaluation.coco_evaluation]: Per-category bbox AP:
| category   | AP     | category   | AP     |
|:-----------|:-------|:-----------|:-------|
| dead       | 80.484 | live       | 72.792 |
```

Figura 4.18: Métricas de bbox para el modelo resnet50 con 2000 iteraciones de entrenamiento.

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.532
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.968
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.564
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.484
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.489
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.565
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.062
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.407
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.583
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.578
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.581
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.607
[06/07 13:01:33 d2.evaluation.coco_evaluation]: Evaluation results for segm:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 53.200 | 96.834 | 56.354 | 48.424 | 48.858 | 56.462 |
[06/07 13:01:33 d2.evaluation.coco_evaluation]: Per-category segm AP:
| category   | AP     | category   | AP     |
|:-----------|:-------|:-----------|:-------|
| dead       | 45.019 | live       | 61.380 |
```

Figura 4.19: Métricas segmentación del modelo resnet50.

Como se indicó en el anterior párrafo, las muestras de coral vivo son detectadas con una precisión bastante alta, en torno al 77 % si lo que deseamos es la media entre los diferentes ***threshold*** (umbrales), en el caso de las inferencias anteriores, se utilizó un umbral de 0.5, siendo la precisión en la detección de los objetos para este umbral del 99.874 % como se puede ver en la métrica AP50. Este baja a 93.464 % si lo que deseásemos sería aumentar el umbral de decisión al 0.75.

4.1.10. Entrenamiento del segundo modelo: maskrcnn + resnet101 para 1000 iteraciones

Tal y como ocurría con el anterior modelo utilizado, volvemos a contar únicamente con dos clases (coral vivo y coral muerto), con estas características el artículo oficial del modelo indica que se pueden conseguir resultados visualmente correctos, tras entrenar durante 1000 iteraciones. En este caso, el entrenamiento duró en total 43 minutos y las pérdidas totales obtenidas a la finalización del mismo fueron 0.989.

```
 eta: 0:04:21  iter: 899  total_loss: 1.055  loss_cls: 0.246  loss_box_reg: 0.420  loss_mask: 0.285  loss_rpn_cls: 0.013
 eta: 0:03:30  iter: 919  total_loss: 1.070  loss_cls: 0.262  loss_box_reg: 0.418  loss_mask: 0.281  loss_rpn_cls: 0.017
 eta: 0:02:38  iter: 939  total_loss: 1.018  loss_cls: 0.222  loss_box_reg: 0.394  loss_mask: 0.281  loss_rpn_cls: 0.017
 eta: 0:01:46  iter: 959  total_loss: 1.072  loss_cls: 0.245  loss_box_reg: 0.403  loss_mask: 0.284  loss_rpn_cls: 0.017
 eta: 0:00:54  iter: 979  total_loss: 1.013  loss_cls: 0.214  loss_box_reg: 0.408  loss_mask: 0.280  loss_rpn_cls: 0.013
 eta: 0:00:02  iter: 999  total_loss: 0.989  loss_cls: 0.240  loss_box_reg: 0.392  loss_mask: 0.270  loss_rpn_cls: 0.018
Overall training speed: 997 iterations in 0:42:32 (2.5606 s / it)
Total training time: 0:42:36 (0:00:03 on hooks)
```

Figura 4.20: Entrenamiento del modelo resnet101 para 1000 iteraciones.

Inferencia del segundo modelo entrenado con 1000 iteraciones de entrenamiento

En las **Figuras 4.21 y 4.22** se puede observar la inferencia del modelo tras un entrenamiento de 1000 iteraciones, comparándolo con el etiquetado manual de uno de los fotogramas del vídeo.

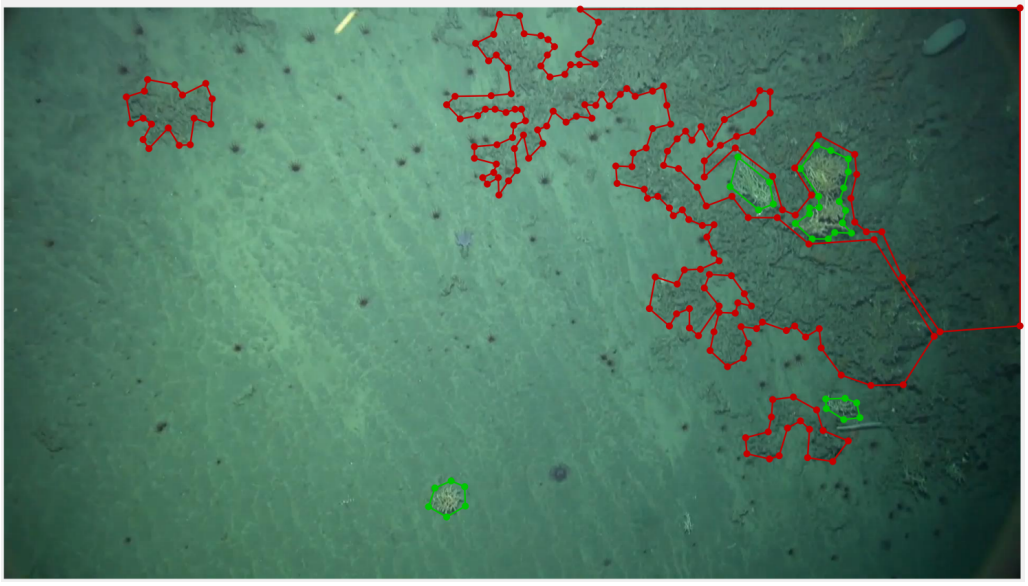


Figura 4.21: Etiquetado de uno de los fotogramas del vídeo con la herramienta Labelme.

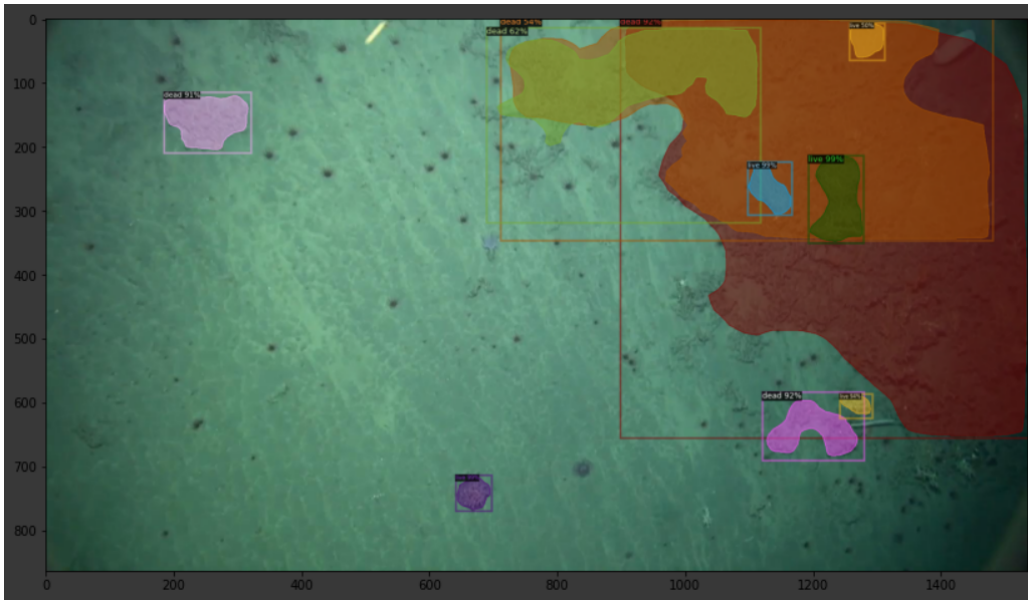


Figura 4.22: Inferencia de la red resnet101 con 1000 iteraciones de entrenamiento.

Otro ejemplo de la inferencia del modelo en otro fotograma diferente se puede observar en las **Figuras** **4.23** y **4.24**:

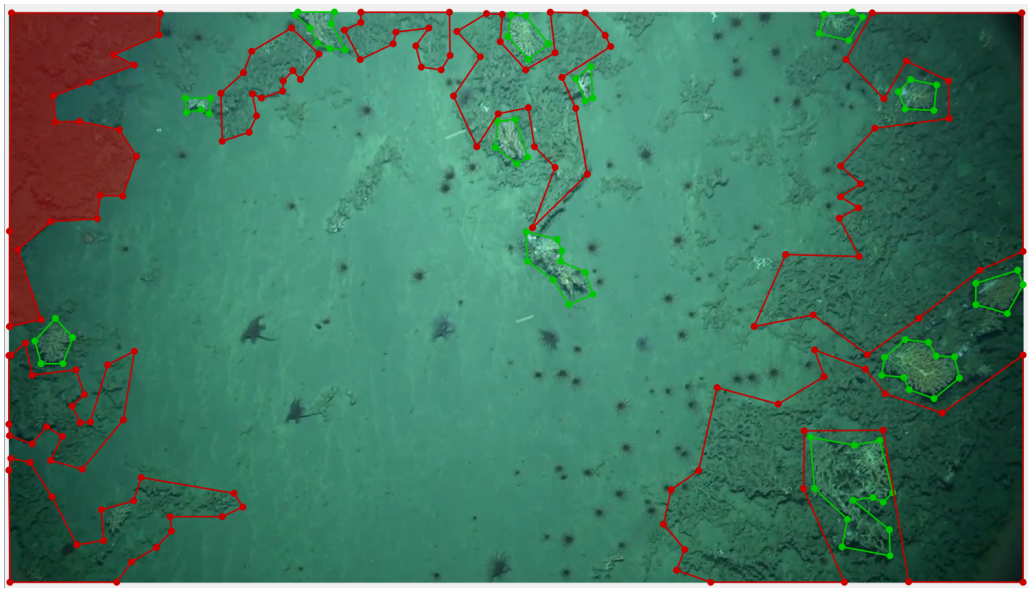


Figura 4.23: Etiquetado de uno de los fotogramas del vídeo con la herramienta Labelme.

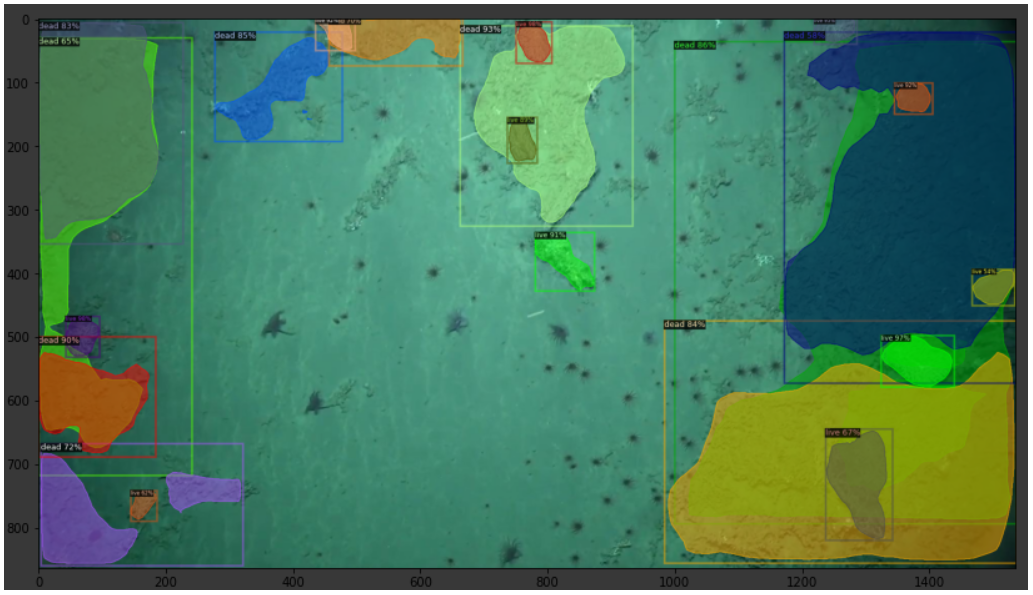


Figura 4.24: Inferencia de la red resnet101 con 1000 iteraciones de entrenamiento.

Métricas tras 1000 iteraciones de entrenamiento

Las métricas obtenidas para el modelo tras 1000 iteraciones se pueden observar en la **Figuras 4.25** y **4.26**.

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.567
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.906
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.605
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.540
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.653
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.591
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.084
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.453
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.661
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.544
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.700
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.690
[06/07 14:17:23 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 56.748 | 90.589 | 60.452 | 53.960 | 65.253 | 59.061 |
[06/07 14:17:23 d2.evaluation.coco_evaluation]: Per-category bbox AP:
| category   | AP     | category   | AP     |
|:-----------|:-------|:-----------|:-------|
| dead       | 52.838 | live       | 60.658 |
```

Figura 4.25: Métricas de la red resnet101 para bbox con 1000 iteraciones.

De igual forma que el anterior modelo entrenado, las métricas para un entrenamiento de únicamente 1000 iteraciones, es bastante pobre, consiguiéndose únicamente un mAP de en torno al 57 % en las bbox para la media de todos los umbrales de decisión. Aunque es cierto y cabe destacar que tal y como pasaba con el anterior modelo, para un umbral de decisión del 0.5 se obtienen los mejores resultados, siendo estos del 90 % de precisión.

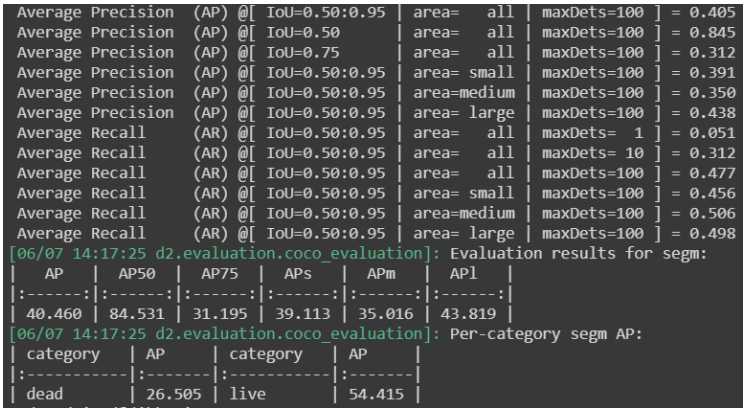


Figura 4.26: Métricas de la red resnet101 para segmentación con 1000 iteraciones.

Para el caso de las métricas para la segmentación de la imagen, ocurre prácticamente lo mismo, con un umbral de decisión de 0.5 se consiguen unas métricas muy buenas, sin embargo, el **mAP** con la media de precisión de todos los umbrales baja al 40 %, quizá algo pobre teniendo en cuenta que se trata del set de validación.

Tal y como se hizo con el anterior modelo, para tratar de mejorar los resultados, se va a aumentar el tiempo de entrenamiento, ya que como se pudo ver en la **Figura 4.20**, las pérdidas totales seguían bajando a cada iteración por lo que es probable que aún tenga margen de mejora.

4.1.11. Entrenamiento del segundo modelo: maskrcnn + resnet101 para 2000 iteraciones

El entrenamiento tuvo en total una duración de una hora y 22 minutos obteniéndose unas pérdidas totales al final del mismo de 0.614.

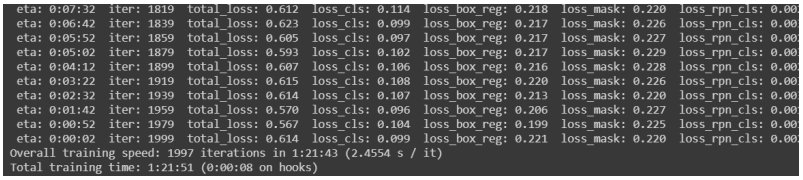


Figura 4.27: Entrenamiento red resnet101 y 2000 iteraciones.

Inferencia del segundo modelo entrenado con 2000 iteraciones de entrenamiento

En las **Figuras 4.28** y **4.29** se puede comparar la inferencia del modelo tras un entrenamiento de 2000 iteraciones, comparándolo con el etiquetado manual de uno de los fotogramas del vídeo.

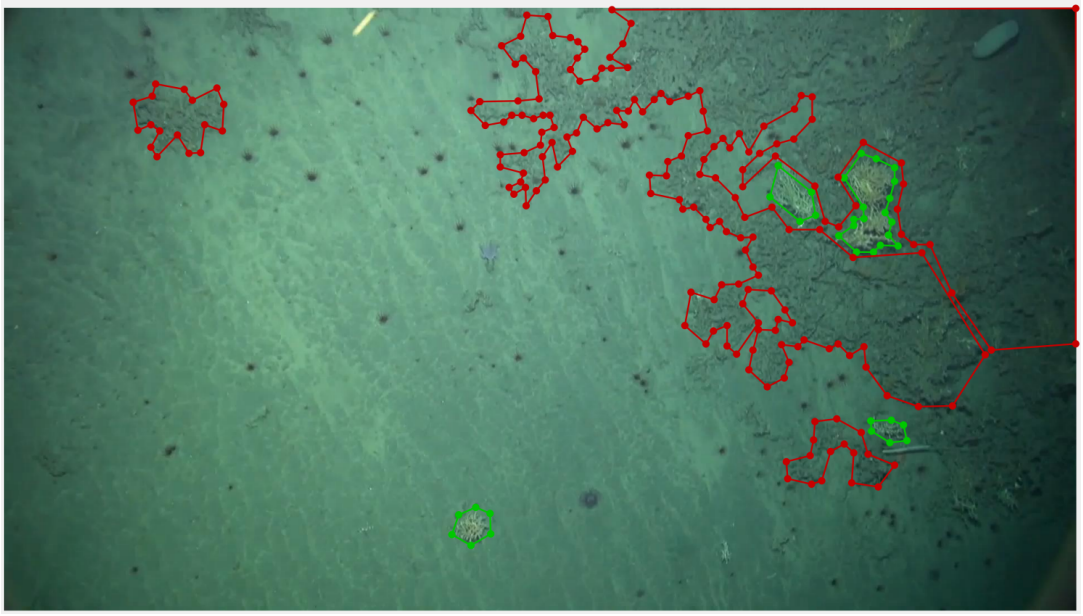


Figura 4.28: Etiquetado de uno de los fotogramas del vídeo con la herramienta Labelme.

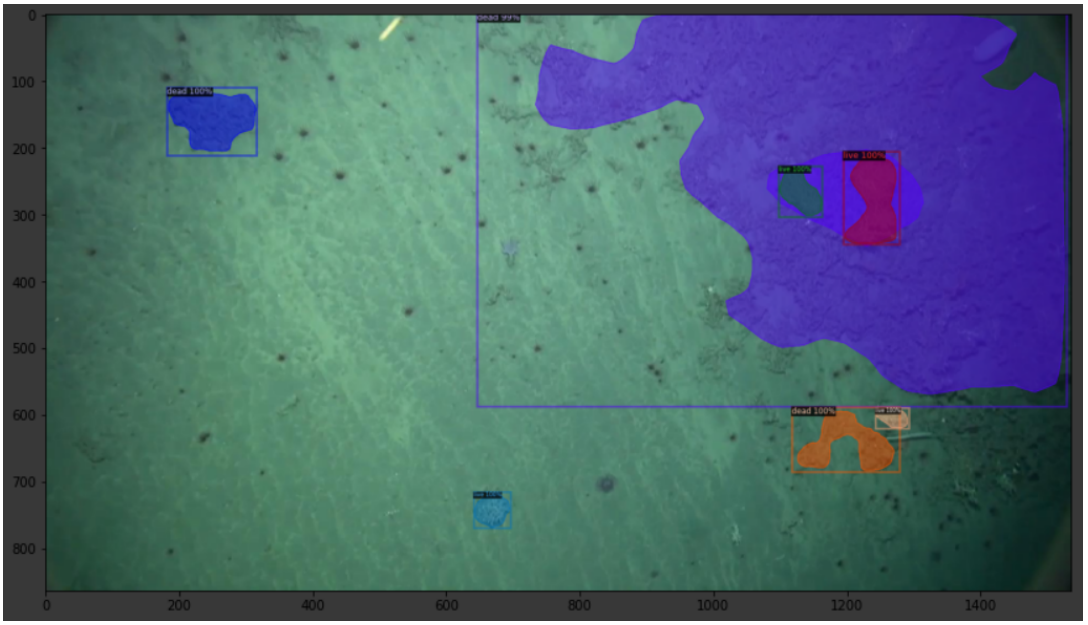


Figura 4.29: Inferencia de la red resnet101 con 2000 iteraciones de entrenamiento.

Otro ejemplo de comparación la inferencia del modelo, con el etiquetado manual en otro fotograma diferente se puede observar en las **Figuras** **4.30** y **4.31**:

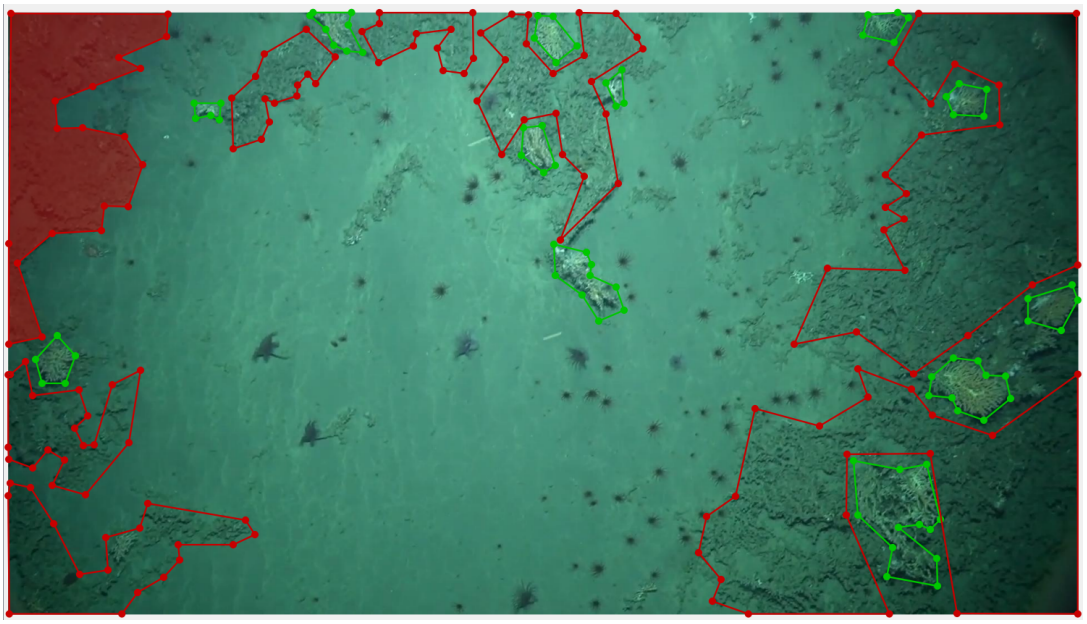


Figura 4.30: Etiquetado de uno de los fotogramas del vídeo con la herramienta Labelme.

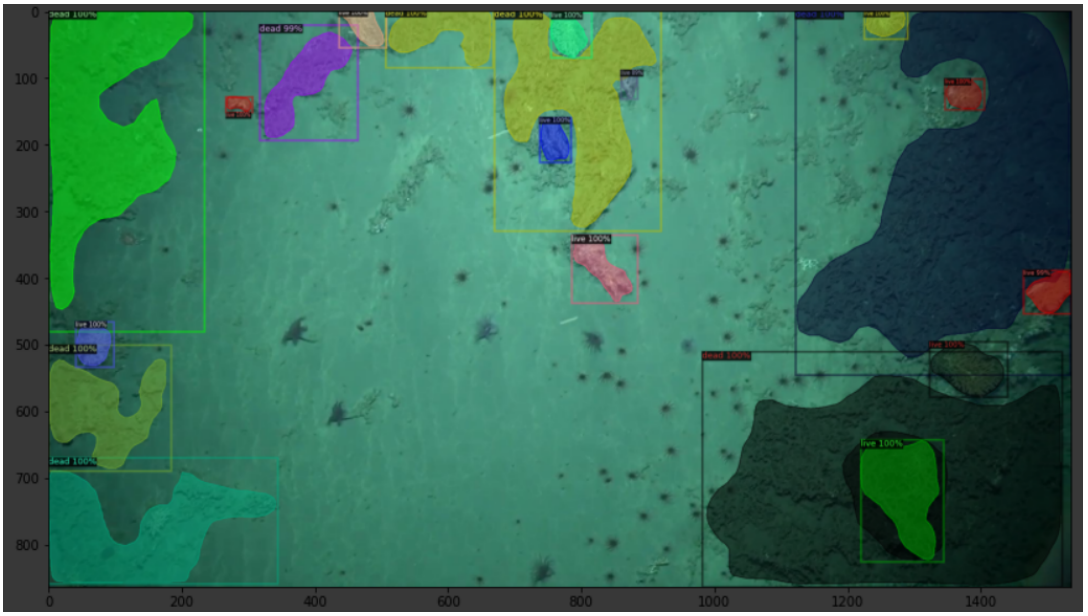


Figura 4.31: Inferencia de la red resnet101 con 2000 iteraciones de entrenamiento.

Métricas tras 2000 iteraciones de entrenamiento

Las métricas obtenidas para el modelo tras 2000 iteraciones se pueden observar en la **Figura 4.32** y **4.33**

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.825
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.965
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.641
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.821
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.841
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.106
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.640
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.863
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.656
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.848
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.891
[06/07 17:14:39 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   |  AP50   |  AP75  |  APs   |  APm   |  APl   |
|:------:|:-------:|:------:|:------:|:------:|:------:|
| 82.507 | 100.000 | 96.480 | 64.139 | 82.074 | 84.150 |
[06/07 17:14:39 d2.evaluation.coco_evaluation]: Per-category bbox AP:
| category   | AP     | category   | AP     |
|:-----------|:-------|:-----------|:-------|
| dead       | 88.425 | live       | 76.589 |
```

Figura 4.32: Métricas de la red resnet101 para detección de objeto con 2000 iteraciones.

Tal y como ocurría con el anterior modelo entrenado, las métricas para un entrenamiento de 2000 iteraciones, mejoran sobremanera a las métricas del mismo modelo con 1000 iteraciones. Se obtiene una precisión sobre las bbox del 82.5 % para la media de todos los umbrales de decisión. Destacan los resultados para el umbral de decisión 0.5, con el cual se obtiene un 100 % de precisión.

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.56
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.98
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.62
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.48
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.52
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.59
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.06
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.43
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.61
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.54
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.60
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.64
[06/07 17:14:39 d2.evaluation.coco_evaluation]: Evaluation results for segm:
|   AP   |  AP50   |  AP75  |  APs   |  APm   |  APl   |
|:------:|:-------:|:------:|:------:|:------:|:------:|
| 56.237 | 98.874 | 62.571 | 48.736 | 52.449 | 59.525 |
[06/07 17:14:39 d2.evaluation.coco_evaluation]: Per-category segm AP:
| category   | AP     | category   | AP     |
|:-----------|:-------|:-----------|:-------|
| dead       | 49.529 | live       | 62.945 |
```

Figura 4.33: Métricas de la red resnet101 para segmentación con 2000 iteraciones.

Para el caso de las métricas de segmentación de imagen, ocurre exactamente lo mismo. Las métricas para la detección del objeto superan por mucho a las de la segmentación de imagen, siendo estas últimas de una media para todos los umbrales de decisión de 56.24 %. De la misma forma que ocurría en el caso de las bbox, cabe destacar un 98.87 %

de precisión para un umbral del 0.5.

4.1.12. Comparación de ambos modelos

En ambos casos el resultado a simple vista es bastante bueno. Teniendo en cuenta el reducido tamaño del *dataset* utilizado, cabía esperar que el entrenamiento no sería tan eficiente. Sin embargo, los resultados son considerados válidos para ambos modelos, llegando a un porcentaje cercano al 80 % en la detección de los objetos e incluso superando este porcentaje en el caso del modelo con resnet 101.

Sin embargo, como cabía de esperar los resultados de la segmentación no son tan óptimos debido a su mayor complejidad. Aunque en ambos casos se obtienen unas métricas por encima del 50 %.

Aunque hay una clara diferencia en la precisión de **bbox** y la precisión a niveles de píxel como se aprecia en las **Tablas** 4.1 y 4.2, con los resultados obtenidos el porcentaje de coral vivo y muerto en un área puede calcularse de forma bastante precisa.

	Tiempo detección una imagen	**mAP bbox**	**mAP segmentación**
resnet50	400ms	50.03	37.35
resnet101	650ms	56.74	40.46

Tabla 4.1: Comparativa entre resnet50 y resnet101 para 1000 iteraciones de entrenamiento

	Tiempo detección una imagen	**mAP bbox**	**mAP segmentación**
resnet50	400ms	76.6	53.2
resnet101	650ms	82.5	56.23

Tabla 4.2: Comparativa entre resnet50 y resnet101 tras 2000 iteraciones de entrenamiento

Además, se destaca que la diferencia entre las dos redes es de aproximadamente un 5 % en favor de la **resnet101**, sin embargo, la velocidad de detección de esta disminuye respecto a la **resnet50**, siendo el tiempo aproximado de detección de una sola imagen con la red **resnet50** de unos 400ms por lo 650ms de la red **resnet101**.

	Entrenamiento de 1000 iteraciones			
	Coral vivo bbox	**Coral muerto bbox**	**Coral vivo Segmentation**	**Coral muerto Segmentation**
resnet50	56.26	43.8	51.5	23.15
resnet101	60.66	52.83	54.4	26.5

Tabla 4.3: Comparativa entre resnet50 y resnet101 precisión para ambas clases para 1000 iteraciones de entrenamiento

	Entrenamiento de 2000 iteraciones			
	Coral vivo bbox	**Coral muerto bbox**	**Coral vivo Segmentation**	**Coral muerto Segmentation**
resnet50	72.79	80.48	61.38	45.02
resnet101	76.59	88.42	62.94	49.52

Tabla 4.4: Comparativa entre resnet50 y resnet101 precisión para ambas clases para 2000 iteraciones de entrenamiento

Además, en las **Tablas 4.3** y **4.4** se observa una tendencia a detectar mejor los corales muertos. Esto, puede deberse a la tendencia del modelo a tener una mayor precisión con los objetos más grandes, que por norma general en este *dataset* son los corales muertos.

4.2. Procedimiento arquitectura U-Net

De nuevo, una de las partes más importantes del proyecto es la selección y el etiquetado de las imágenes que formarán parte del futuro ***dataset*** de trabajo.

Para esta nueva arquitectura, el ***dataset*** se creó en base a un vídeo con resolución 1080x1920 prestado por el **IEO**. Este, fue separado en los diferentes fotogramas que componían el vídeo gracias a un script de **Matlab** y se fueron seleccionando dichos fotogramas con varios saltos en el tiempo, obteniendo así un set de datos lo suficientemente amplio para que nuestra red no vea siempre la misma forma, color y textura del coral. En esta ocasión, el set de datos fue más pequeño que el utilizado en la anterior arquitectura, ya que en el artículo del autor del modelo se asegura que la necesidad de datos por parte de **U-Net** es la más baja frente a la competencia.

4.2.1. Dataset utilizado

En esta ocasión, el entrenamiento se llevó a cabo únicamente con 7 fotogramas, etiquetados con la herramienta de etiquetado en línea **Supervisely**[10], cuya interfaz se puede ver en la **Figura 4.34**:

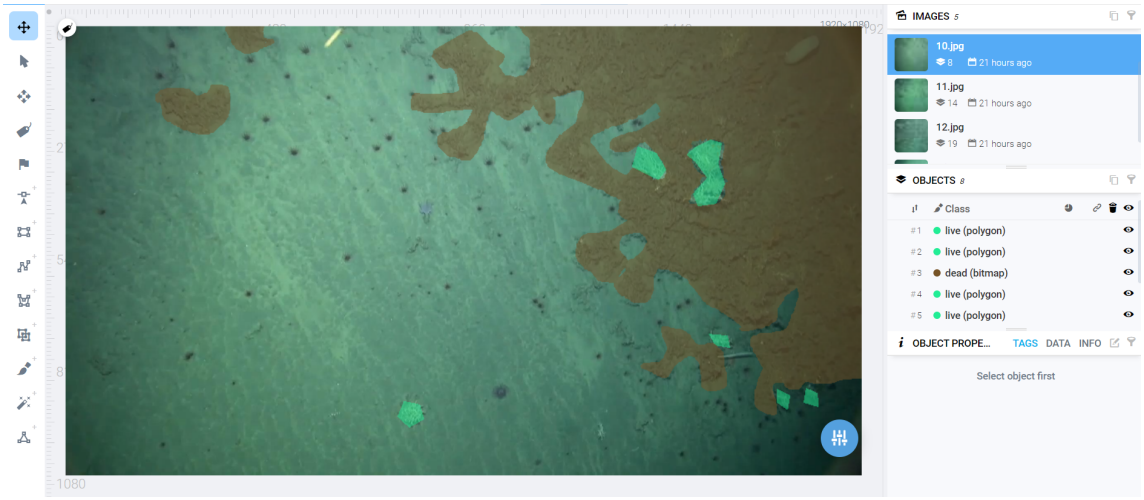


Figura 4.34: Etiquetado de uno de los fotogramas del vídeo con la herramienta Supervisely.

Como se puede ver, se han declarado dos clases, el color marrón abarca todo el coral muerto, mientras que el color verde abarca el coral vivo.

A la salida de esta herramienta se puede trabajar con la elección de descarga entre:

- Anotaciones
- Imagen original + anotaciones
- Imagen original + anotaciones + máscaras

Siendo las anotaciones un archivo con formato JSON con la siguiente forma:


```
{
    "id": 465564269,
    "classId": 1546401,
    "description": "",
    "geometryType": "polygon",
    "labelerLogin": "SergioSierra",
    "createdAt": "2020-06-16T15:34:32.903Z",
    "updatedAt": "2020-06-16T15:36:41.461Z",
    "tags": [],
    "classTitle": "live",
    "points": {
        "exterior": [
            [
                48,
                964
            ],
            [
                68,
                989
            ],
            [
                116,
                956
            ],
            [
                110,
                926
            ]
        ],
        "interior": []
    }
},
```


Figura 4.35: Etiquetado de uno de los fotogramas del vídeo con la herramienta Supervisely.

Tras el etiquetado se debe obtener una máscara, formato PNG, ya que es la forma en la que la red U-net recibe la entrada. Esta visualmente es una máscara completamente negra, sin embargo, en la **Figura 4.36** se han variado los colores en escala de grises diferenciando cada clase.

Figura 4.36: Ejemplo de máscaras con colores ficticios en escala de grises.

Este tipo de máscaras trabajan con la escala de valores RGB para los colores siendo el *background* el negro absoluto, es decir, (0,0,0) por defecto, mientras que las distintas clases

etiquetadas irán tomando otros valores otorgados por nosotros con el siguiente código:

```
{
  "action": "data",
  "src": [
    "dataset_coral_unet/ds"
  ],
  "dst": "$0",
  "settings": {
    "classes_mapping": "default"
  }
},
{
  "action": "save_masks",
  "src": [
    "$0"
  ],
  "dst": "dataset_coral_unet",
  "settings": {
    "masks_machine": true,
    "masks_human": true,
    "gt_machine_color": {
      "dead": [
        1,
        1,
        1
      ],
      "live": [
        2,
        2,
        2
      ]
    },
```

Figura 4.37: Etiquetado de uno de los fotogramas del vídeo con la herramienta Supervisely.

Como se puede ver, el color RGB (1,1,1) se le asigna al coral muerto y (2,2,2) para el coral vivo. Es por eso que las diferencias de tono no se pueden apreciar a simple vista en la imagen de la máscara.

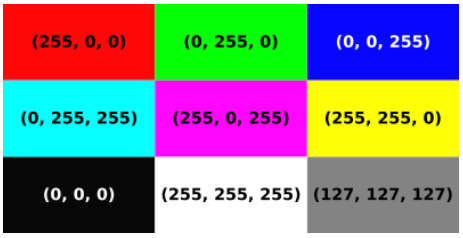


Figura 4.38: Gamas de colores según sus valores RGB.

Como se mencionó anteriormente, U-net se destaca por conseguir buenos resultados sin la necesidad de trabajar con un gran *dataset*. Por lo tanto, solo se han etiquetado 5 fotogramas de un vídeo con resolución de 1080x1920 aplicando un script de aumento de datos[21] sobre el etiquetado.

4.2.2. Bibliotecas utilizadas

La biblioteca utilizada en este caso es **Keras**, un framework de alto nivel para el aprendizaje, escrito en Python y capaz de correr sobre los frameworks TensorFlow, CNTK, o Theano.

Fue desarrollado con el objeto de facilitar un proceso de experimentación rápida. Sus fuertes se centran en ser amigable para el usuario, modular y extensible.

4.2.3. Entrenamiento

El entrenamiento tuvo una duración aproximada de media hora para las 5 épocas programadas, las pérdidas totales al final del mismo de 0.0385.

```
model.train(
    train_images =  "/content/drive/My Drive/tfm/prueba_unet_supervisely/img",
    train_annotations = "/content/drive/My Drive/tfm/prueba_unet_supervisely/masks",
    checkpoints_path = "/content/drive/My Drive/tfm/unet_semantic_coral/last_weights/vgg_unet_1" , epochs=5
)
```

```
 40%|████▍      | 2/5 [00:00<00:00, 15.76it/s]Verifying training dataset
100%|███████████| 5/5 [00:00<00:00, 16.17it/s]
Dataset verified!
Epoch 1/5
512/512 [==============================] - 347s 678ms/step - loss: 0.5339 - accuracy: 0.8841
saved  /content/drive/My Drive/tfm/unet_semantic_coral/last_weights/vgg_unet_1.0
Epoch 2/5
512/512 [==============================] - 326s 637ms/step - loss: 0.1072 - accuracy: 0.9626
saved  /content/drive/My Drive/tfm/unet_semantic_coral/last_weights/vgg_unet_1.1
Epoch 3/5
512/512 [==============================] - 325s 635ms/step - loss: 0.0365 - accuracy: 0.9871
saved  /content/drive/My Drive/tfm/unet_semantic_coral/last_weights/vgg_unet_1.2
Epoch 4/5
512/512 [==============================] - 327s 638ms/step - loss: 0.0338 - accuracy: 0.9893
saved  /content/drive/My Drive/tfm/unet_semantic_coral/last_weights/vgg_unet_1.3
Epoch 5/5
512/512 [==============================] - 325s 635ms/step - loss: 0.0385 - accuracy: 0.9887
saved  /content/drive/My Drive/tfm/unet_semantic_coral/last_weights/vgg_unet_1.4
```

Figura 4.39: Entrenamiento de la red U-Net.

4.2.4. Resultados

Tras el entrenamiento se hicieron varias pruebas tanto para el set de validación como para el formato de test. Los resultados fueron los siguientes:

	Tiempo de detección	**Precisión**
VGG_Unet	1.21 segundos	98.8 %

Tabla 4.5: Métricas modelo U-Net con transfer learning sobre los pesos base VGG.

Por tener una comparativa entre ambas arquitecturas, se dispondrán los resultados de las dos imágenes de validación utilizadas en el modelo detectron2.

Los resultados fueron los siguientes:

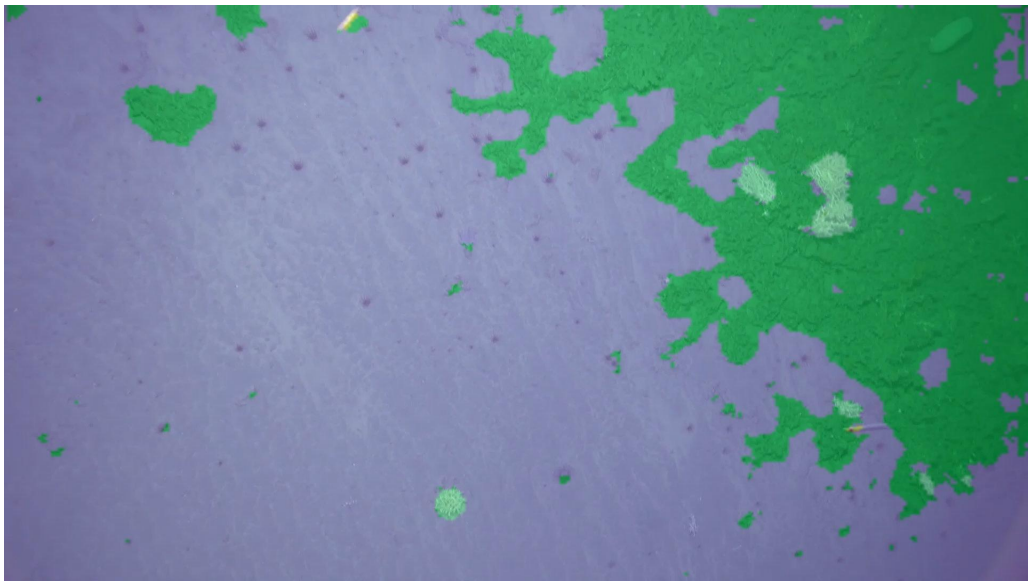


Figura 4.40: Ejemplo de inferencia de la red U-Net.

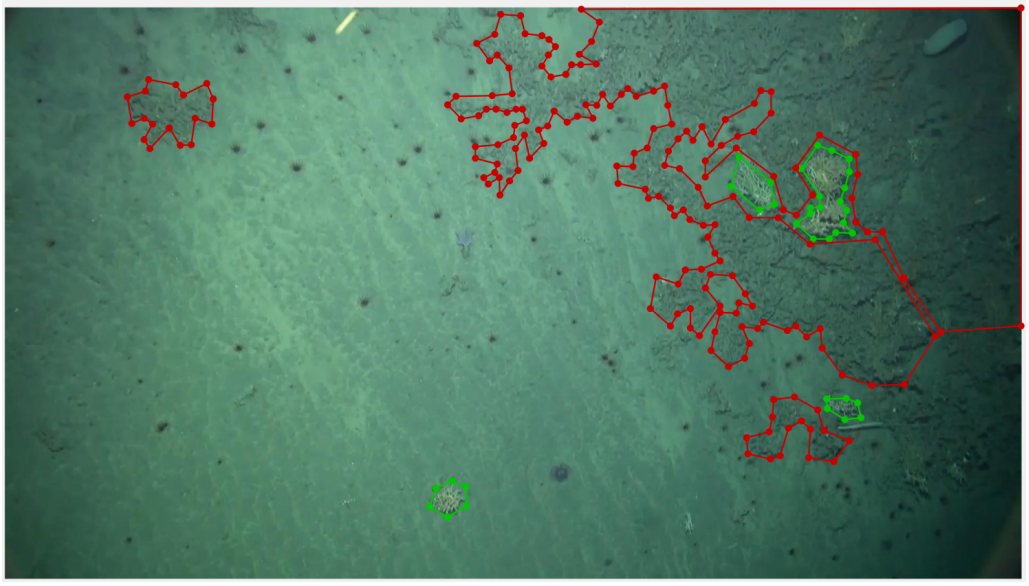


Figura 4.41: Etiquetado de uno de los fotogramas del vídeo con la herramienta Supervisely.

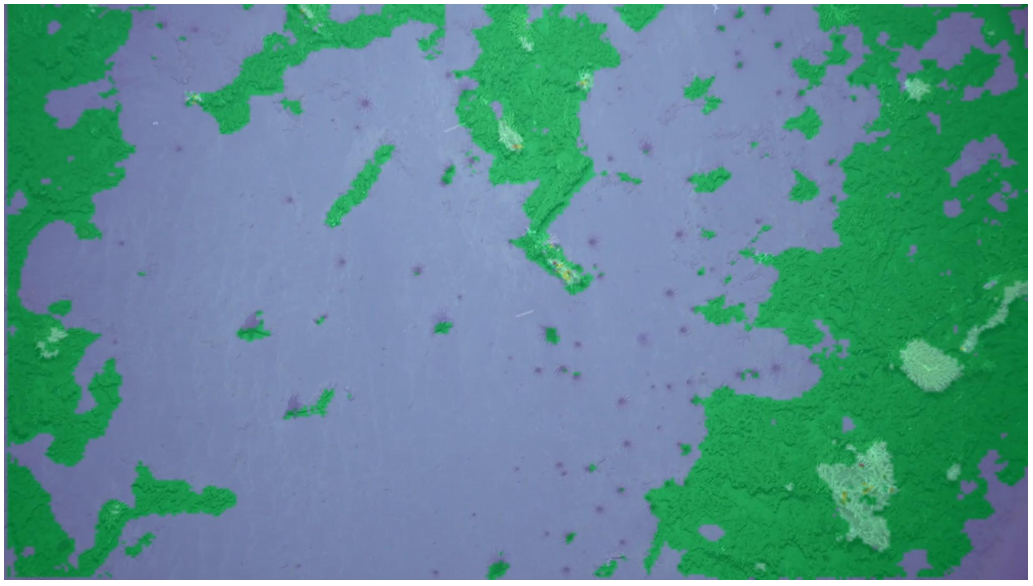


Figura 4.42: Ejemplo de inferencia de la red U-Net.

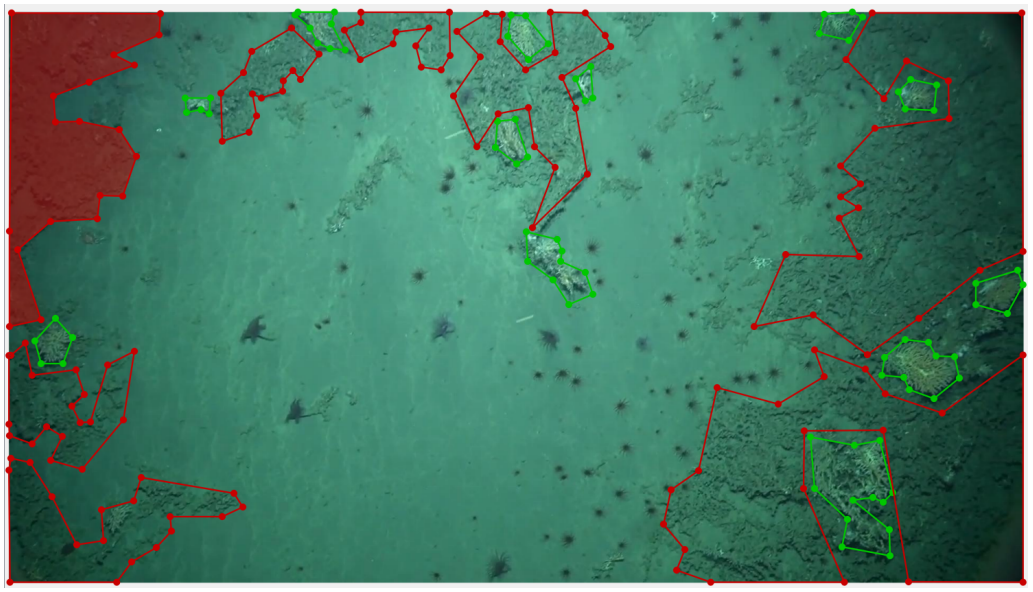


Figura 4.43: Etiquetado de uno de los fotogramas del vídeo con la herramienta Supervisely.

Por último, un ejemplo de la entrada que se le pasa al modelo para la inferencia del mismo y la máscara de salida obtenida para dicha imagen.

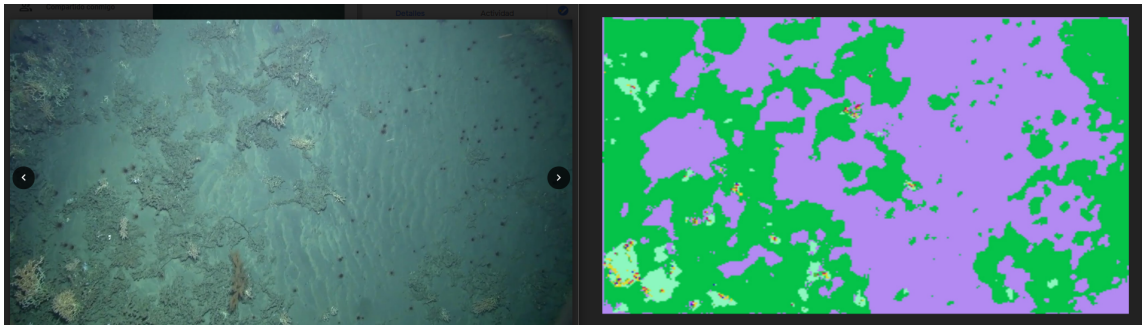


Figura 4.44: Ejemplo de inferencia de la red U-Net.

Capítulo 5

Detección de objetos

Esta fase del trabajo, consiste en la detección de tres especies de los fondos marinos que se encuentran con frecuencia en el hábitat del cañón submarino de Avilés. Dichas especies son:

- **Artemisinas** (Artemisina transiens)
- **Phakellias**(Phakellia ventilabrum)
- **Dendrophillias** (D. cornigera)

En la **Figura** 5.1 se puede ver la forma y características de cada una de las anteriores especies.

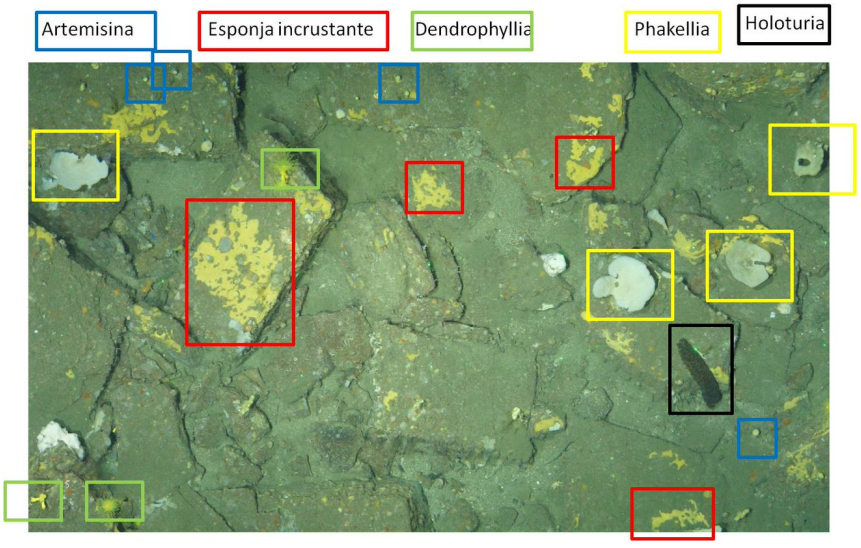


Figura 5.1: Clasificación de las especies más frecuentes en el hábitat marino de los cañones de Avilés. *Imagen proporcionada por el* ***IEO***

Los cuadros azules corresponden a las artemisinas, los verdes a las dendrophillias y los amarillos a las phakellias.

5.1. Dataset

El set de datos, como en todo el proyecto, fue sustentado por el IEO, consta de vídeos y miles de imágenes que fueron etiquetados manualmente por expertos de la institución con la herramienta de etiquetado online Supervisely, obteniéndose un set de datos para el entrenamiento de 3252 imágenes en la que se incluyen objetos de las 3 diferentes clases con una proporción equitativa.

Figura 5.2: Imagen del dataset

```
- <size>
    <width name="width">416</width>
    <height name="height">416</height>
    <depth name="depth">3</depth>
  </size>
  <segmented name="segmented">0</segmented>
- <object>
    <name name="name">Dendrophyllia</name>
    <pose name="pose">Unspecified</pose>
    <truncated name="truncated">0</truncated>
    <difficult name="difficult">0</difficult>
  - <bndbox>
      <xmin name="xmin">160</xmin>
      <ymin name="ymin">202</ymin>
      <xmax name="xmax">326</xmax>
      <ymax name="ymax">327</ymax>
    </bndbox>
  </object>
- <object>
    <name name="name">Artemisina</name>
    <pose name="pose">Unspecified</pose>
    <truncated name="truncated">0</truncated>
    <difficult name="difficult">0</difficult>
  - <bndbox>
      <xmin name="xmin">251</xmin>
      <ymin name="ymin">7</ymin>
      <xmax name="xmax">300</xmax>
      <ymax name="ymax">62</ymax>
    </bndbox>
  </object>
```

Figura 5.3: Etiquetado de la imagen

En la **Figura 5.2** se puede ver una de las imágenes del *dataset* en la que se pueden diferenciar claramente varios objetos. Por ejemplo, una dendropyhllia, y varias artemisinas. En la **Figura 5.3** se detalla la posición de los objetos dentro de la imagen, en dicha figura hay dos ejemplos de estos objetos.

5.2. Data Augmentation

5.3. Modelo utilizado: YOLOv4

El modelo utilizado con el que se obtuvieron las mejores prestaciones, fue el modelo **YOLOv4** sustentado por la red de detección darknet[1][19]. Se eligió este modelo debido a sus altas puntuaciones frente a sus rivales en cuanto a precisión, pero sobre todo por la gran velocidad de detección característica de ellos. Se probaron varias versiones de YOLO, en

primer lugar, se utilizó **YOLOv2** sobre Matlab y aunque los resultados fueron buenos, no tuvieron nada que hacer contra la **YOLOv3** y **YOLOv4**, los cuales son modelos más modernos que tienen diversas mejoras en cada una de sus capas, haciéndolos modelos mucho mejores tanto en precisión como en tiempo de detección, tal y como dicen sus creadores. En la siguiente gráfica se puede apreciar dichas diferencias:

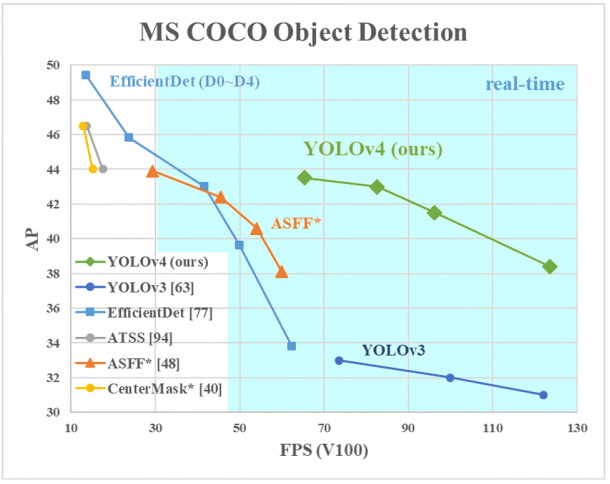


Figura 5.4: Comparación entre precisión y velocidad de inferencia de algunos modelos frente a YOLOv4.*Imagen de* ***Synced***

Esta gráfica está sustentada sobre la precisión media obtenida por diferentes modelos infiriendo sobre el *dataset* de COCO en el eje de las Y y los fotogramas que es capaz de detectar en un segundo en el eje de las X. Con esto, vemos que el modelo **YOLOv4** sobre Darknet[1] obtiene un **mAP** (precisión media con los diferentes umbrales de decisión) de entre 41 y 44 %, una de las más altas de la gráfica. Mientras que es el único modelo que con una precisión aceptable es capaz de inferir sobre un vídeo en tiempo real, y que es capaz de trabajar por encima de los **30 fps**.

5.4. Preparación del set de datos para el modelo YOLOv4

Tal y como se mencionó anteriormente, el *dataset* fue etiquetado de manera manual con la herramienta supervisely. Esta herramienta produce a la salida un archivo formato .json con las coordenadas de la caja en la que se encuentra el objeto a clasificar sobre la imagen. Sin embargo, tanto las imágenes como las etiquetas deben prepararse de una forma específica para el correcto funcionamiento del modelo. Para esto se hace uso de los códigos presentados en los **Anexos** **B.1** (convierte todos los archivos de etiquetado a un único csv) y **B.2** (cambia el formato del csv anterior a formato YOLO para poder trabajar con el modelo posteriormente). Obteniéndose las coordenadas de cada objeto con la siguiente forma:

```
0  0.173077  0.444712  0.091346  0.105769
0  0.631010  0.449519  0.074519  0.115385
0  0.533654  0.580529  0.100962  0.112981
```

Figura 5.5: Formato YOLO de las coordenadas de una caja en la imagen.

En la **Figura 5.5** se puede ver un ejemplo de una imagen en la que se ha etiquetado 3 objetos, siendo los 3 de la clase 0 (phakellia) y sus 4 coordenadas representativas dentro de la imagen en el formato YOLO.

Además, se debe obtener un archivo con formato .txt de la siguiente forma:

```
obj/fdyzrj_TV26_0010_frame00281_004657_vf.jpg
obj/mqpyav_TV26_0010_frame00281_004658_br.jpg
obj/aexoso_TV26_0010_frame00281_004628_hf.jpg
obj/exfphd_TV26_0010_frame00281_004677_br.jpg
obj/xkdjqm_TV26_0010_frame00281_004634_oo.jpg
obj/wgwrup_TV26_0010_frame00281_004641_oo.jpg
obj/fnkaio_TV26_0010_frame00281_004680_oo.jpg
obj/gtcfpt_TV26_0010_frame00281_004660_vf.jpg
obj/uyedmh_TV26_0010_frame00281_004647_br.jpg
obj/ojpwfw_TV26_0010_frame00281_004642_hf.jpg
obj/upljqp_TV26_0010_frame00281_004643_hf.jpg
obj/bueequ_TV26_0010_frame00281_004655_oo.jpg
obj/sunosa_TV26_0010_frame00281_004643_vf.jpg
obj/ysvoem_TV26_0010_frame00281_004679_hf.jpg
obj/nlyxeq_TV26_0010_frame00281_004658_vf.jpg
obj/npuqie_TV26_0010_frame00281_004636_oo.jpg
obj/ucjwgk_TV26_0010_frame00281_004645_hf.jpg
obj/bwqjql_TV26_0010_frame00281_004650_br.jpg
```

Figura 5.6: Formato YOLO de las coordenadas de una caja en la imagen.

En el que se detalla el directorio en el que se encuentran las imágenes del set de datos.

Además hay que escribir otro archivo, de nombre **obj.names** en el que se detalle el nombre de las clases que hay en el set de datos.

```
Phakellia
Artemisina
Dendrophyllia
```

Figura 5.7: obj.names

Finalmente, un archivo de nombre **obj.data** en el que se detalle tanto el número de clases, como los directorios de los *dataset* de entrenamiento y validación, como un directorio de *backup* en el que ser irán guardando los pesos cada 1000 iteraciones de entrenamiento, pudiendo escoger para la inferencia unos pesos anteriores a los actuales en caso de que se consideren mejores sus métricas.

```
classes = 3
train = data/train.txt
valid = data/test.txt
names = data/obj.names
backup = /mydrive/yolov3/backup
```

Figura 5.8: obj.data

5.5. Entrenamiento

El entrenamiento tuvo una duración de alrededor de 5000 iteraciones las cuales como se ve en la **Figura 5.9** se llevaron a cabo en 3 etapas. Se obtuvo el mejor **mAP** entorno a las 4000 iteraciones para el set de validación consiguiendo unas perdidas tras el entrenamiento en torno a 0.7.

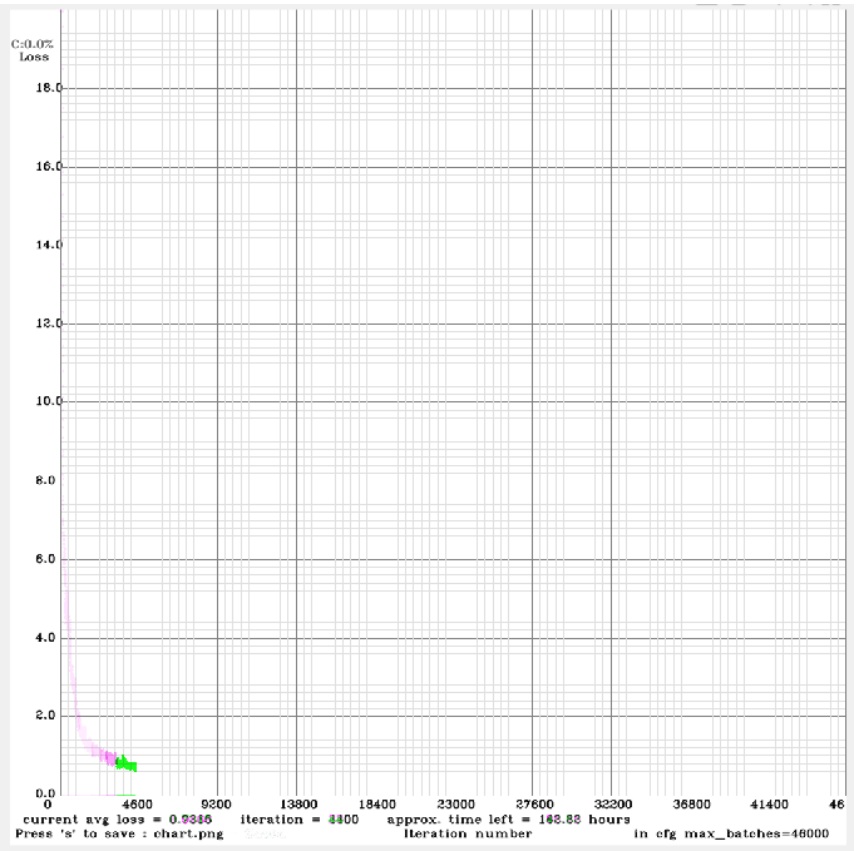


Figura 5.9: Perdidas del entrenamiento.

5.6. Resultados

Una vez obtenidos los pesos del entrenamiento con mejor **mAP** sobre el set de validación se pasó al testeo del modelo sobre un ortomosaico como el de la **Figura 5.10**. Un ortomosaico

es una composición de muchas imágenes georeferenciadas que son alineadas para formar un modelo 3D del fondo marino en este caso. La vista de ese modelo desde la vertical constituye el ortomosaico, que en este caso ha sido generado con el *software* **Pix4D** a partir de las imágenes del *dataset*.

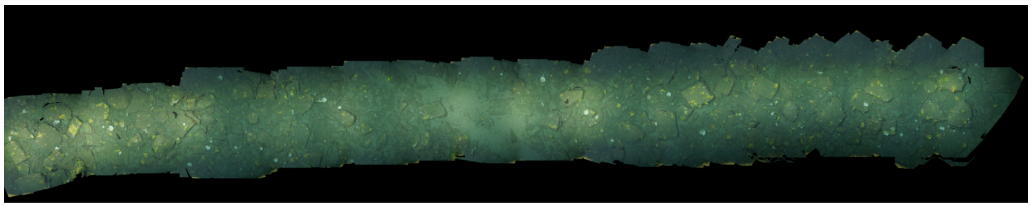


Figura 5.10: Ortomosaico de ejemplo con el que se comprobará la precisión del modelo.

Este tiene unas dimensiones de 23238x4891, por lo que para obtener mejores resultados sobre él, se ha troceado en imágenes 418x418 (resolución nativa de la red).

Una vez inferido sobre el ortomosaico de test las métricas obtenidas fueron las siguientes:

```
ruth classes boxes count:  {'Dendrophyllia': 213, 'Phakellia': 62, 'Artemisina': 132}
Predicted classes boxes count:  {'Phakellia': 65, 'Dendrophyllia': 254, 'Artemisina': 94}
Confusion matrix
predict=          Phakellia       Dendrophyllia   Artemisina      other
truth= Dendrophyllia      0       161     0       52
truth= Phakellia          51      0       1       10
truth= Artemisina         0       0       70      62
truth= other      14      93      23      0

TP:  {'Dendrophyllia': 161, 'Phakellia': 51, 'Artemisina': 70}
FP:  {'Dendrophyllia': 0, 'Phakellia': 1, 'Artemisina': 0, 'other': 130}
TN:  {'other': 0}
FN:  {'Dendrophyllia': 52, 'Phakellia': 10, 'Artemisina': 62}
Precision for  Dendrophyllia  1.00
Recall for     Dendrophyllia  0.76
f1-score for   Dendrophyllia  0.86
Precision for  Phakellia  0.98
Recall for     Phakellia  0.84
f1-score for   Phakellia  0.90
Precision for  Artemisina  1.00
Recall for     Artemisina  0.53
f1-score for   Artemisina  0.69
mean Precision   0.99
mean Recall      0.71
mean f1-score    0.82
```

Figura 5.11: Métricas de inferencia del modelo para cada clase.

Tras estudiar las métricas detenidamente, se puede apreciar que aunque la precisión para las tres clases es muy buena, hay una clase que es especialmente difícil de detectar para el modelo, la clase artemisina. Este problema se puede achacar al reducido tamaño de esta especie.

En las siguientes imágenes se puede apreciar un pequeño montaje de dos imágenes de tamaño **1080x1920** cada una, habiendo sido inferidas anteriormente por el modelo con un **umbral de decisión del 0.3**, es decir, para dar por válido una detección el modelo

tiene que estar al menos seguro al 30 % de su decisión.

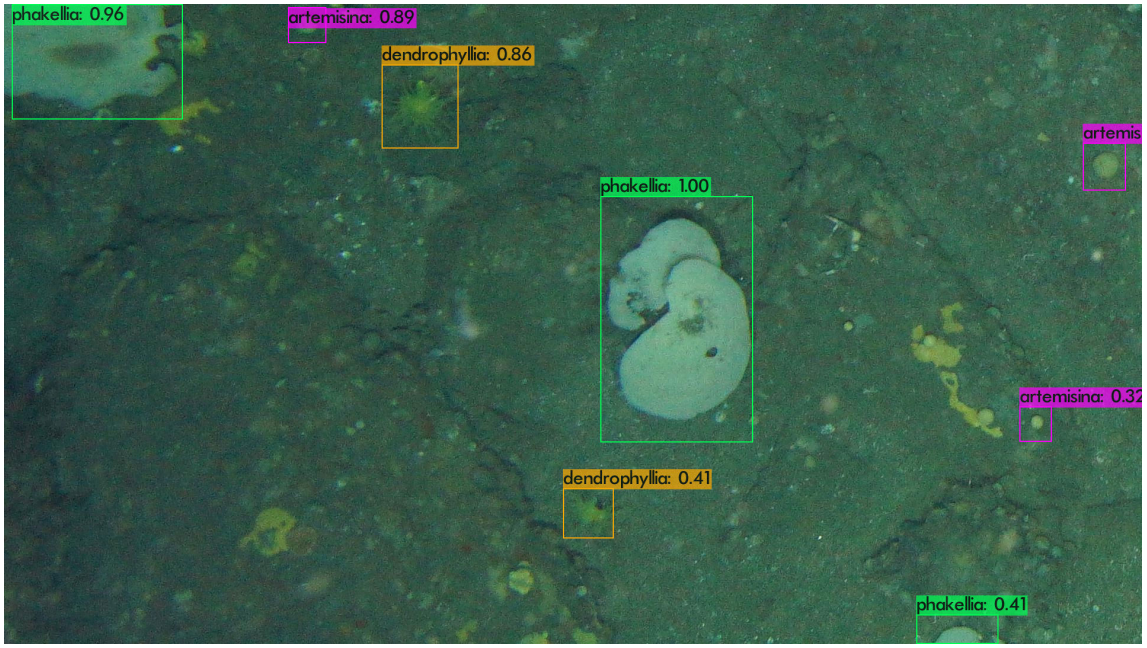


Figura 5.12: Ejemplo de inferencia del modelo.

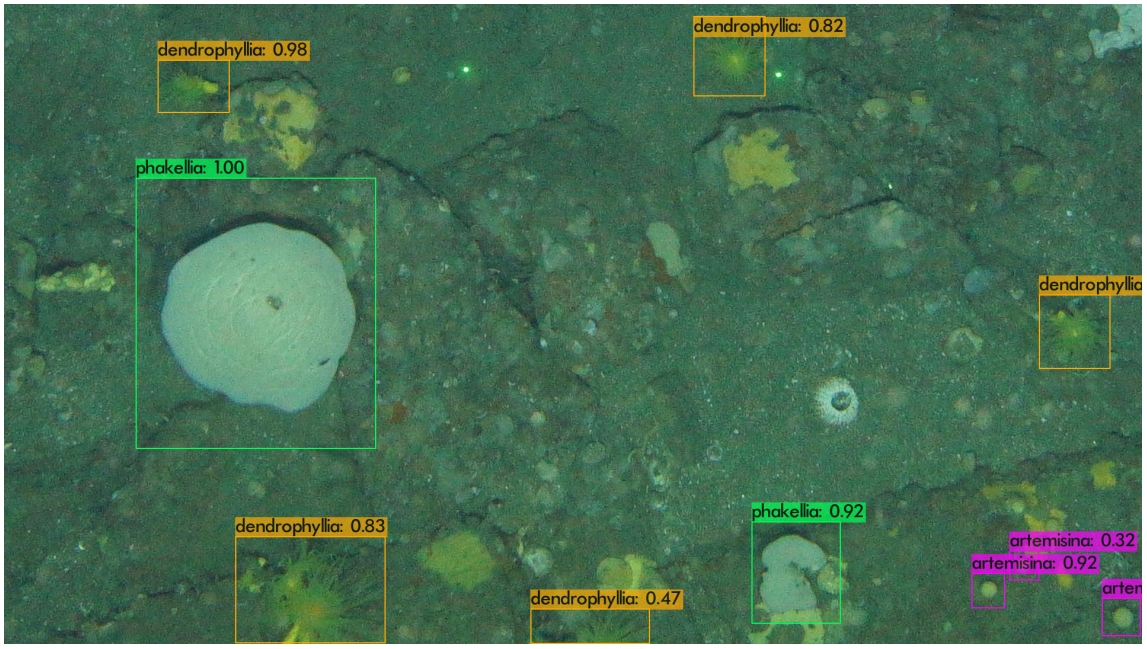


Figura 5.13: Ejemplo de inferencia del modelo.

Capítulo 6

Conclusiones

Con todo lo anterior, se ha conseguido abarcar las dos secciones del ***Deep Learning*** en cuanto a reconocimiento de imagen a nivel de píxel que se explicaban en el **Punto 2.5.1**. La herramienta **Detectron2** a través de la arquitectura ResNet nos permite detectar y clasificar los píxeles de lo que la máquina considera un objeto, es decir, **segmentación de instancias** ya que cada objeto detectado tendrá un color exclusivo. En segundo lugar, con la arquitectura U-Net basada en el modelo VGG cada objeto perteneciente a una clase será clasificado con el mismo color que los demás de su misma condición, esto se denomina segmentación semántica.

Cabe destacar la gran diferencia entre el primer modelo entrenado y el segundo en cuanto a la detección de una sola imagen, siendo estos los siguientes resultados:

	Tiempo de detección	**Precisión a nivel de píxel(%)**
ResNet50	0.4 segundos	53.2
ResNet101	0.65 segundos	56.23
VGG_Unet	1.21 segundos	98.8

Tabla 6.1: Tiempos de detección y precisión de ambos modelos para el set de validación

En la anterior tabla comparativa, se puede ver como la **ResNet50** tiene un tiempo de detección cercano a la tercera parte del modelo con mayor precisión (U-Net), sin embargo, tanto este como su hermano mayor, la **ResNet101** sacrifican parte de la precisión para ser más rápidos. Sin embargo, los tres algoritmos no serían lo suficientemente rápidos como para actuar en tiempo real a no ser que se tratase de un vídeo con una tasa de fotogramas por segundo extremadamente baja.

Por lo tanto, un método sencillo para hacer la inferencia sobre un vídeo es utilizar los códigos del **Anexo A.2**, superponiendo el fotograma original a la máscara obtenida de la

inferencia y juntando todas las inferencia en un vídeo con el **Anexo A.3**.

Finalmente, en el **Tabla 6.2** se pueden ver los resultados obtenidos por el modelo YOLOv4, se consideran métricas muy buenas, sobre todo teniendo en cuenta que el **F1-score** de dos de las especies está por encima del 80 %, sumándole la dificultad que conlleva para el propio ser humano diferenciar las **esponjas incrustantes amarillas** de la **Figura 5.1**. Los peores resultados, son claramente los obtenidos para las artemisas, que tal y como se mencionó en el **Punto 5.6**, se puede deber a la pérdida de información al cambiar a la resolución nativa de la red al ser objetos que ocupan tan pocos píxeles de la imagen.

	Precision	**Recall**	**F1-score**
Dendrophyllia	100 %	76 %	86 %
Phakellia	98 %	84 %	90 %
Artemisina	100 %	53 %	69 %

Tabla 6.2: Precision, Recall y F1-score para las 3 especies con el modelo YOLOv4.

Capítulo 7

Líneas futuras

Este trabajo puede funcionar como base para futuras mejoras por parte del **IEO**, por ejemplo, el estudio del desarrollo de distintas áreas de los cañones de Avilés con el paso del tiempo u otras áreas distintas de las que se quiera hacer un estudio temporal.

Una de las mejoras a tener en cuenta, sería el aumento del set de datos por parte de expertos en etiquetado de especies de corales, incluso se podrían etiquetar nuevas clases que diferencien entre las **distintas especies de corales**[21], teniendo en cuenta así más información a parte de si el coral está vivo o muerto.

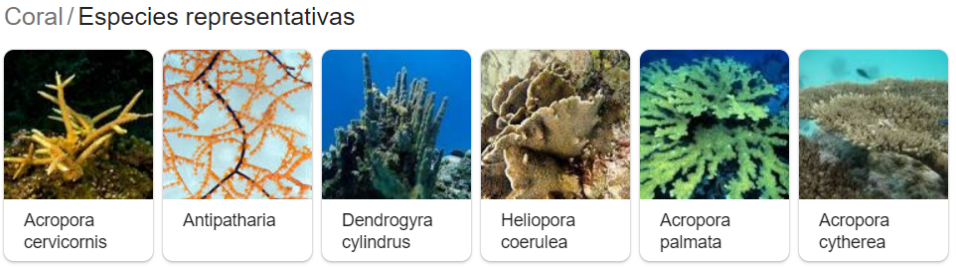


Figura 7.1: Distintas especies de corales. *Recuperado de* ***Oceana***

Con una simple búsqueda en Google de las especies de corales existentes, nos encontramos con las especies de la **Figura 7.1**.

Otra de las posibles ideas a tener en cuenta en el futuro para mejorar el modelo, es un posible equilibrio entre las clases del *dataset*. Con el *dataset* actual, aunque el coral muerto es más abundante en extensión, es mucho menor en cuanto a número, al ser el coral vivo mucho más esporádico. Tal y como se menciona en [32] el "desbalanceo de clases" en segmentación semántica lleva al modelo a etiquetar píxeles de la clase mayoritaria.

Por último, se podría generar un *script* que teniendo en cuenta los punteros láser que se

pueden ver en varias de las imágenes del *dataset* como se puede ver en la **Figura 7.2** fuese capaz de interpretar el área real que abarca una imagen.

Una idea para este objetivo sería: conociendo la distancia física real de dichos punteros (20cm) y los píxeles que ocupa en el eje X el rectángulo correspondiente a la detección. Extrapolar esa distancia conocida a la resolución de la imagen en cuestión, conociendo así de forma aproximada el área abarcada.

Figura 7.2: Ejemplo de dos puntos para toma de medida real a escala.

Para el caso de la detección de objetos, se podría aumentar el número de especies a tratar, en este caso únicamente se tuvo en cuenta solo tres especies. Sin embargo, en el set de datos se pueden encontrar hasta 7 especies diferentes.

Otro aspecto es la mejora de las métricas, especialmente en la clase artemisina, que es la clase con las métricas menos competentes para este modelo.

Apéndice A

ANEXO segmentación de imagen

En este anexo, se deja constancia de los códigos y notebooks utilizados para el desarrollo del proyecto.

- **Labelme2coco.py** script utilizado para la conversión del *dataset* al formato utilizado por detectron2.
- **original_mask.m** este script se utiliza para superponer las imágenes originales con las obtenidas a la salida del modelo U-Net (máscaras)
- **frames2video.m** este script se utiliza para convertir en vídeo la salida del script original_mask.m
- **video2frames.m** este script se utiliza para convertir en las imágenes de entrenamiento los fotogramas del vídeo.

A.1. Labelme2coco.py

```
1  import os
2  import argparse
3  import json
4
5  from labelme import utils
6  import numpy as np
7  import glob
8  import PIL.Image
9
10
11 class labelme2coco(object):
12     def __init__(self, labelme_json=[], save_json_path="./coco.json"):
13         """
14         :param labelme_json: the list of all labelme json file paths
15         :param save_json_path: the path to save new json
16         """
17         self.labelme_json = labelme_json
18         self.save_json_path = save_json_path
```

```
        self.images = []
        self.categories = []
        self.annotations = []
        self.label = []
        self.annID = 1
        self.height = 0
        self.width = 0

        self.save_json()

    def data_transfer(self):
        for num, json_file in enumerate(self.labelme_json):
            with open(json_file, "r") as fp:
                data = json.load(fp)
                self.images.append(self.image(data, num))
                for shapes in data["shapes"]:
                    label = shapes["label"].split("_")
                    if label not in self.label:
                        self.label.append(label)
                    points = shapes["points"]
                    self.annotations.append(self.annotation(points, label, num))
                    self.annID += 1

        # Sort all text labels so they are in the same order across data splits.
        self.label.sort()
        for label in self.label:
            self.categories.append(self.category(label))
        for annotation in self.annotations:
            annotation["category_id"] = self.getcatid(annotation["category_id"])

    def image(self, data, num):
        image = {}
        img = utils.img_b64_to_arr(data["imageData"])
        height, width = img.shape[:2]
        img = None
        image["height"] = height
        image["width"] = width
        image["id"] = num
        image["file_name"] = data["imagePath"].split("/")[-1]

        self.height = height
        self.width = width

        return image

    def category(self, label):
        category = {}
        category["supercategory"] = label[0]
        category["id"] = len(self.categories)
        category["name"] = label[0]
        return category

    def annotation(self, points, label, num):
        annotation = {}
        contour = np.array(points)
        x = contour[:, 0]
        y = contour[:, 1]
        area = 0.5 * np.abs(np.dot(x, np.roll(y, 1)) - np.dot(y, np.roll(x, 1)))
        annotation["segmentation"] = [list(np.asarray(points).flatten())]
        annotation["iscrowd"] = 0
        annotation["area"] = area
```

```
80          annotation["image_id"] = num
81
82          annotation["bbox"] = list(map(float, self.getbbox(points)))
83
84          annotation["category_id"] = label[0]  # self.getcatid(label)
85          annotation["id"] = self.annID
86          return annotation
87
88      def getcatid(self, label):
89          for category in self.categories:
90              if label == category["name"]:
91                  return category["id"]
92          print("label: {} not in categories: {}.".format(label, self.categories))
93          exit()
94          return -1
95
96      def getbbox(self, points):
97          polygons = points
98          mask = self.polygons_to_mask([self.height, self.width], polygons)
99          return self.mask2box(mask)
100
101     def mask2box(self, mask):
102
103         index = np.argwhere(mask == 1)
104         rows = index[:, 0]
105         clos = index[:, 1]
106
107         left_top_r = np.min(rows)  # y
108         left_top_c = np.min(clos)  # x
109
110         right_bottom_r = np.max(rows)
111         right_bottom_c = np.max(clos)
112
113         return [
114             left_top_c,
115             left_top_r,
116             right_bottom_c - left_top_c,
117             right_bottom_r - left_top_r,
118         ]
119
120     def polygons_to_mask(self, img_shape, polygons):
121         mask = np.zeros(img_shape, dtype=np.uint8)
122         mask = PIL.Image.fromarray(mask)
123         xy = list(map(tuple, polygons))
124         PIL.ImageDraw.Draw(mask).polygon(xy=xy, outline=1, fill=1)
125         mask = np.array(mask, dtype=bool)
126         return mask
127
128     def data2coco(self):
129         data_coco = {}
130         data_coco["images"] = self.images
131         data_coco["categories"] = self.categories
132         data_coco["annotations"] = self.annotations
133         return data_coco
134
135     def save_json(self):
136         print("save coco json")
137         self.data_transfer()
138         self.data_coco = self.data2coco()
139
140         print(self.save_json_path)
```

```
        os.makedirs(
            os.path.dirname(os.path.abspath(self.save_json_path)), exist_ok=True
        )
        json.dump(self.data_coco, open(self.save_json_path, "w"), indent=4)


if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(
        description="labelme annotation to coco data json file."
    )
    parser.add_argument(
        "labelme_images",
        help="Directory to labelme images and annotation json files.",
        type=str,
    )
    parser.add_argument(
        "--output", help="Output json file path.", default="trainval.json"
    )
    args = parser.parse_args()
    labelme_json = glob.glob(os.path.join(args.labelme_images, "*.json"))
    labelme2coco(labelme_json, args.output)
```

A.2. original_mask.m

```
workingDir="C:\Users\sergi\Desktop\mask_imagen_memoria\images"

imageNames = dir(fullfile(workingDir,'*.jpg'));
imageNames = {imageNames.name}';

workingDir1="C:\Users\sergi\Desktop\mask_imagen_memoria\masks"
maskNames = dir(fullfile(workingDir1,'*.jpg'));
maskNames = {maskNames.name}';

workingDir2="C:\Users\sergi\Desktop\mask_imagen_memoria\result"

for ii = 1:length(imageNames)
    img = imread(fullfile(workingDir,imageNames{ii}));
    mask = imread(fullfile(workingDir1,maskNames{ii}));
    alpha(0)
    %mask_trans= alpha(mask, 0.3)
    imshow(mask)
    mezcla = imfuse(img,mask,'blend','Scaling','joint');
    imshow(mezcla)
    filename = [sprintf('%03d',ii) '.jpg'];
    fullname = fullfile(workingDir2,'images',filename);
    imwrite(mezcla,fullname)
end
```

A.3. frames2video.m

```
workingDir="C:\Users\sergi\Desktop\mezcla_video\images\"
imageNames = dir(fullfile(workingDir,'*.jpg'));
imageNames = {imageNames.name}';

outputVideo = VideoWriter(fullfile(workingDir,'coral_mascaras_original.avi'));
```

```
6
7  open(outputVideo)
8
9  for ii = 1:length(imageNames)
10     img = imread(fullfile(workingDir,imageNames{ii}));
11     writeVideo(outputVideo,img)
12 end
13 close(outputVideo)
```

A.4. video2frames.m

```
1  workingDir = tempname;
2  mkdir(workingDir,'images')
3
4  cd C:\Users\sergi\Desktop\tfm\dataset_coral
5  shuttleVideo = VideoReader('coral_0a15.mp4');
6
7  ii = 1;
8  while hasFrame(shuttleVideo)
9  img = readFrame(shuttleVideo);
10 filename = [sprintf('%03d',ii) '.jpg'];
11 fullname = fullfile(workingDir,'images',filename);
12 imwrite(img,fullname)    % Write out to a JPEG file (img1.jpg, img2.jpg, etc.)
13 ii = ii+1;
14 end
```

Apéndice B

ANEXO Detección de objetos

En este anexo, se deja constancia de los códigos y notebooks utilizados para el desarrollo del proyecto de detección de objetos.

- **convertir_csv.py** script utilizado para la conversión del *dataset* etiquetado a un solo archivo formato csv con las coordenadas de cada bbox.
- **csv_to_yolo** este script se utiliza para convertir el archivo csv del anterior script a formato YOLO (un txt por imagen).

B.1. convertir_csv.py

```
1  from __future__ import division, print_function, absolute_import
2
3  import pandas as pd
4  import numpy as np
5  import csv
6  import re
7  import cv2
8  import os
9  import glob
10 import xml.etree.ElementTree as ET
11
12 import io
13
14 from PIL import Image
15 from collections import namedtuple, OrderedDict
16
17 import shutil
18 import urllib.request
19 import tarfile
20
21
22 #adjusted from: https://github.com/datitran/raccoon_dataset
23
24 #converts the annotations/labels into one csv file for each training and testing
       labels
```

```
25 #creats label_map.pbtxt file
26
27 dataset_path = 'd:/Universidad de Cantabria/Proyecto deepRAMP - Documentos/dataset/
       Aviles_IA418/circarock1_608px_3clasesPAD_withblanks'
28
29 images_extension = 'jpg'
30
31 # takes the path of a directory that contains xml files and converts
32 #  them to one csv file.
33
34 # returns a csv file that contains: image name, width, height, class, xmin, ymin,
       xmax, ymax.
35 # note: if the xml file contains more than one box/label, it will create more than
       one row for the same image. each row contains the info for an individual box.
36 def xml_to_csv(path):
37   classes_names = []
38   xml_list = []
39
40   for xml_file in glob.glob(path + '/*.xml'):
41     tree = ET.parse(xml_file)
42     root = tree.getroot()
43     for member in root.findall('object'):
44       classes_names.append(member[0].text)
45       value = (root.find('filename').text,
46                int(root.find('size')[0].text),
47                int(root.find('size')[1].text),
48                member[0].text,
49                int(member[4][0].text),
50                int(member[4][1].text),
51                int(member[4][2].text),
52                int(member[4][3].text))
53       xml_list.append(value)
54   column_name = ['filename', 'width', 'height', 'class', 'xmin', 'ymin', 'xmax', '
       ymax']
55   xml_df = pd.DataFrame(xml_list, columns=column_name)
56   classes_names = list(set(classes_names))
57   classes_names.sort()
58   return xml_df, classes_names
59
60 # for both the train_labels and test_labels csv files, it runs the xml_to_csv()
       above.
61 label_path = os.path.join(dataset_path,'ann')
62 image_path = os.path.join(os.getcwd(), label_path)
63 xml_df, classes = xml_to_csv(label_path)
64 xml_df.to_csv(f'{label_path}.csv', index=None)
65 print(f'Successfully converted {label_path} xml to csv.')
66
67 # Creating the `label_map.pbtxt` file
68 label_map_path = os.path.join(dataset_path,"label_map.pbtxt")
69
70 pbtxt_content = ""
71
72 #creats a pbtxt file the has the class names.
73 for i, class_name in enumerate(classes):
74     # display_name is optional.
75     pbtxt_content = (
76         pbtxt_content
77         + "item {{\n    id: {0}\n    name: '{1}'\n    display_name: 'Gun'\n }}\n\n
              ".format(i + 1, class_name)
78     )
79 pbtxt_content = pbtxt_content.strip()
```

```
80 with open(label_map_path, "w") as f:
81     f.write(pbtxt_content)
```

B.2. csv_to_yolo.py

```
1
2  import pandas as pd
3  import os
4  import numpy as np
5  import re
6
7  #Get the content of the csv to process the data from the config file
8
9  content_dct={}
10 with open('config.txt') as f:
11     content=f.readlines()
12     for i in content:
13         var,val = i.split('=')
14         content_dct[var.strip()]=val.strip()
15 print(content_dct)
16
17
18 flag=0
19
20 #load the labels as dictionary
21
22 label_dct={}
23 for ind,lab in enumerate(content_dct['class_name'].split(',')):
24     label_dct[lab]=ind
25 print(label_dct)
26
27 #Read the Dataframe
28 try:
29     int(content_dct['width'])
30     int(content_dct['height'])
31     df= pd.read_csv(content_dct['file_name'], sep=",",dtype={content_dct['xmin']:
           float,
32                                                              content_dct['ymin']:
                                                                   float,content_dct[
                                                                   'xmax']:float,
33                                                              content_dct['ymax']:
                                                                   float})
34     flag=1
35 except:
36     df= pd.read_csv(content_dct['file_name'], sep=",",dtype={content_dct['width']:
           int,content_dct['height']:int,
37                                                 content_dct['xmin']:float,content_dct
                                                       ['ymin']:float,
38                                                 content_dct['xmax']:float,content_dct
                                                       ['ymax']:float})
39 df.head()
40
41 new_df=df.copy(deep=True)
42 new_df.head()
43
44
45 #Map the labels to integer
46 try:
47     new_df=new_df.loc[new_df['class'].isin(label_dct.keys())]
48     new_df['class']=new_df[content_dct['label']].map(label_dct)
49     print(new_df.head())
50 except:
51     print('The column label mentioned in the config file doesn\'t exist in the csv
           file')
```

```
52      raise
53
54
55  #converting the VOC points into yolo points
56  if(flag==0):
57      try:
58          new_df['x']=((df[content_dct['xmin']]+df[content_dct['xmax']])/2)/df[
                content_dct['width']]
59          new_df['y']=((df[content_dct['ymin']]+df[content_dct['ymax']])/2)/df[
                content_dct['height']]
60          new_df['w']=(df[content_dct['xmax']]-df[content_dct['xmin']])/df[
                content_dct['width']]
61          new_df['h']=(df[content_dct['ymax']]-df[content_dct['ymin']])/df[
                content_dct['height']]
62      except:
63          print('The column mentioned in the config file doesn\'t exist in the csv
                file')
64          raise
65  else:
66      new_df['x']=((df[content_dct['xmin']]+df[content_dct['xmax']])/2)/int(
            content_dct['width'])
67      new_df['y']=((df[content_dct['ymin']]+df[content_dct['ymax']])/2)/int(
            content_dct['height'])
68      new_df['w']=(df[content_dct['xmax']]-df[content_dct['xmin']])/int(content_dct['
            width'])
69      new_df['h']=(df[content_dct['ymax']]-df[content_dct['ymin']])/int(content_dct['
            height'])
70  new_df.head()
71
72
73  final_df=pd.DataFrame({'filename':new_df['filename'],'label':new_df['class'],'x':
        new_df['x'],'y':new_df['y'],'w':new_df['w'],'h':new_df['h']})
74  final_df.head()
75
76  unique_image=final_df['filename'].unique()
77
78  count = 0
79  for i in unique_image:
80      if count % 100 == 0:
81         print(count)
82      count = count +1
83      file_name=i.split('.')[0]+'.txt'
84      row_series=final_df.loc[final_df['filename']==i,'label':'h']
85      try:
86          with open(os.path.join(content_dct['txt_file_path'],file_name),'w') as f:
87              row_series.to_string(f,header=False,index=False)
88      except:
89          print('please check the txt file path')
90          raise
91
92
93  #writting a txt file with all the images path
94  try:
95      with open(os.path.join(content_dct['txt_images_path'],'../train.txt'),'w') as f
            :
96          count = 0
97          for i in unique_image:
98              if count % 100 == 0:
99                  print(count)
100             count = count +1
101             f.write(i+'\n') # le he quitado el prefijo 'train/'
```

```
except:
    print('please check the image names file path')
    raise


#saving the csv with yolo data for future use
final_df.to_csv(content_dct['output_csv_name'],index=False)
```

Bibliografía y referencias

[1] Alexey. AlexeyAB/Darknet. 2016. 2020. GitHub, https://github.com/AlexeyAB/darknet.

[2] Biodiversidad, Fundación. "Cañón de Avilés - INDEMARES-AVILÉS 0412 - IEO." Indemares, 5 Aug. 2015, https://www.indemares.es/proyecto/canon-de-aviles-indemares-aviles-0412-ieo.

[3] Castleman kr 1996. "Digital image processing", prentice-hall, englewood cliffs, new jersey 07632.

[4] H. li, k. ota and m. dong, "Learning iot in edge: Deep learning for the internet of things with edge computing, Inieee network,Vol. 32, no. 1, pp. 96-101, jan.-feb. 2018.

[5] Heras, Jose Martinez. "Precision, Recall, F1, Accuracy en clasificación." IArtificial.net, 17 Nov. 2019, https://iartificial.net/precision-recall-f1-accuracy-en-clasificacion/.

[6] Khandelwal, Renu. "Computer Vision: Instance Segmentation with Mask R-CNN." Medium, 27 Nov. 2019, https://towardsdatascience.com/computer-vision-instance-segmentation-with-mask-r-cnn-7983502fcad1.

[7] Michelucci, umberto. advanced applied deep learning: Convolutional neural networks and object detection.apress, 2019.

[8] Mwiti, Derrick. "A 2019 Guide to Semantic Segmentation." Medium, 23 June 2020, https://heartbeat.fritz.ai/a-2019-guide-to-semantic-segmentation-ca8242f5a7fc.

[9] Sanz, Óscar Martín de la Fuente. "Google Colab: Python y Machine Learning en la nube." Adictos al trabajo, 4 June 2019, https://www.adictosaltrabajo.com/2019/06/04/google-colab-python-y-machine-learning-en-la-nube/.

[10] Shorten, Connor. "Introduction to ResNets." Medium, 15 May 2019, https://towardsdatascience.com/introduction-to-resnets-c0a830a288a4.

[11] Shrivastav, Namratesh. "Confusion Matric(TPR,FPR,FNR,TNR), Precision, Recall, F1-Score." Medium, 20 Jan. 2020, https://medium.com/datadriveninvestor/confusion-matric-tpr-fpr-fnr-tnr-precision-recall-f1-score-73efa162a25f.

[12] Stalin Jara. Técnicas de Segmentación de Imágenes.

https://es.slideshare.net/stalinheavy/tcnicas-de-segmentacin-de-imgenes: :text=Resumen %3A %20La %20segmentaci %C3 %B3n %20de %20im %C3 %A1genes,la %20comprensi %C3 %B3n %20de %20im %C3 %A1genes %20o.

[13] Supervisely - Web Platform for Computer Vision. Annotation, Training and Deploy. https://supervise.ly/videos/.

[14] Tanner, Gilbert. "Detectron2 Train a Instance Segmentation Model." Gilbert Tanner, https://gilberttanner.com/blog/detectron2-train-a-instance-segmentation-model.

[15] Team, keras. keras documentation: Keras api reference. https://keras.io/api/.

[16] Tensorflow core, tensorflow, 2020. available: https://www.tensorflow.org/guide?hl=es.

[17] Wada, kentaro. wkentaro/labelme. 2016. 2020. github, https://github.com/wkentaro/labelme.

[18] "Deep Learning Framework Power Scores 2018 • CODESIGN.BLOG." CODESIGN.BLOG, 22 Sept. 2018, https://codesign.blog/2018/09/22/deep-learning-framework-power-scores-2018/.

[19] "Entendiendo la Segmentación Semántica con UNET - - Aprendizaje Automático - 2020." Sciencewal, https://es.sciencewal.com/28944-understanding-semantic-segmentation-with-unet-6be4f42d4b47-42.

[20] "Guía básica para entender los algoritmos de Machine Learning." dheybicervan, 15 Mar. 2020, https://dheybicervan.com/guia-basica-para-entender-los-algoritmos-de-machine-learning/.

[21] "TIPOS DE CORALES - ¡Nombres de las especies con imágenes!" expertoanimal.com, https://www.expertoanimal.com/tipos-de-corales-24271.html.

[22] "¿Qué es el Deep Learning?" SmartPanel, 10 Apr. 2018, https://www.smartpanel.com/que-es-deep-learning/.

[23] I. Arel, D. C. Rose, and T. P. Karnowski. Deep machine learning - a new frontier in artificial intelligenceresearch [research frontier].IEEE Computational Intelligence Magazine, 5(4):13–18, 2010.

[24] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy ofobject detection, 2020.

[25] Raul E. Lopez Briega. Redes neuronales convolucionales con tensorflow. 2016.

[26] Jason Brownlee. "A gentle introduction to transfer learning for deep learning", machine learning mastery, 2017.

[27] Carles Ventura Royo. Universitat Oberta de Catalunya. La segmentación sem ántica y sus benchmarks. 2016.

[28] E.soria and A.Blanco. "Redes neuronales artificiales", acta. pp. 25-33, 2020.

[29] Rui Ma, Pin Tao, and Huiyun Tang. Optimizing data augmentation for semantic segmentation on small-scaledataset. pages 77–81, 06 2019.

[30] Umberto. Michelucci. Advanced applied deep learning: Convolutional neural networks and object detection.1st ed. edition, apress, 2019.

[31] Carlos Garcia Moreno. "¿Qué es el deep learning y para qué sirve?",Indra.Blog neo, innovacion+tecnologia.

[32] "Machine Learning - What Is Imbalance in Image Segmentation?" Stack Overflow, https://stackoverflow.com/questions/45914214/what-is-imbalance-in-image-segmentation.

[33] Hui, Jonathan. "YOLOv4." Medium, 4 May 2020, https://medium.com/@jonathan_hui/yolov4-c9901eaa8e61.

[34] Redmon, Joseph, et al. "You Only Look Once: Unified, Real-Time Object Detection." ArXiv:1506.02640 [Cs], May 2016. arXiv.org, http://arxiv.org/abs/1506.02640.

[35] Mejorando la tecnología del vehículo «Politolana» del IEO-Santander – eDrónica. https://edronica.com/2017/06/01/mejorando-la-tecnologia-del-vehiculo-politolana-del-ieo-santander/.

[36] Gonzalez, Jesus Enrique Carranza. Instituto Español de Oceanografía - Centro Oceanográfico de Santander. ESPAÑA. http://www.ieo-santander.net/.

[37] "La segmentación semántica y sus benchmarks." Informatica ++, 26 May 2016, http://informatica.blogs.uoc.edu/2016/05/26/la-segmentacion-semantica-y-sus-benchmarks/.