UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Detección en tiempo real del punto de agarre del objeto más accesible en entornos industriales mediante aprendizaje profundo.

Trabajo Fin de Máster

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

AUTOR/A: Aparicio Alonso, Eduardo

Tutor/a: Gómez Adrian, Jon Ander

Cotutor/a: Blanes Noguera, Juan Francisco

CURSO ACADÉMICO: 2021/2022

Departament de Sistemes Informàtics i Computació

Universitat Politècnica de València

Detección en tiempo real del punto de agarre del objeto más accesible en entornos industriales mediante aprendizaje profundo

Trabajo Fin de Máster

Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Autor: Aparicio Alonso, Eduardo

Tutor: Gómez Adrián, Jon Ander

Cotutor: Blanes Noguera, Juan Francisco

2021/2022

Resumen

En este proyecto se pretende obtener un sistema basado en redes neuronales que, en tiempo real, sea capaz de detectar el mejor agarre robótico del objeto más accesible entre los detectados en ese instante en un entorno industrial, concretamente para tareas de carga-descarga en máquinas, *picking* u otras tareas de precisión. Esto se va a lograr mediante redes neuronales usadas en segmentación semántica entrenadas con un conjunto de entrenamiento creado a partir de imágenes RGB y su codificación de la profundidad, las cuales son extraídas desde la cámara Intel RealSense Depth Camera D435 usada en el entorno de trabajo y cuyo etiquetado de segmentación se ha realizado mediante la herramienta CVAT. La red se encargará de reconocer la segmentación de los objetos disponibles para ser agarrados en ese instante (sin ser superpuestos por otros objetos), y con esta salida y mediante técnicas de visión por computador con la librería OpenCV de Python, se obtendrá el mejor punto de agarre posible para uno de ellos.

Palabras clave: Robotics, Deep Learning, Computer Vision, Machine Learning

Abstract

This project aims to obtain a system based on neural networks that, in real-time, can detect the best robotic grasp of the most accessible object among those detected at that moment in an industrial environment, specifically in tasks of machine loading-unloading, picking, and other precision tasks. This will be achieved by means of neural networks for semantic segmentation trained with a training set created from RGB images and their depth encoding, which are extracted from the environment's camera Intel RealSense Depth Camera D435 and whose segmentation labelling has been carried out using the CVAT tool. This net will recognize the segmentation of the objects available to be grasped at that moment (without opposition from other objects), and with this output and through computer vision techniques implemented in the OpenCV library in Python, the best possible grasp of one of the objects will be obtained.

Keywords: Robotics, Deep Learning, Computer Vision, Machine Learning

Agradecimientos

Han sido varias las personas que, ya sea académica o moralmente, me han proporcionado su apoyo en la realización de este trabajo y, como es debido, he de expresarles mi gratitud.

En primer lugar, me gustaría agradecer tanto a mi tutor Jon Ander Gómez Adrián como a mi cotutor Juan Francisco Blanes Noguera por todo su esfuerzo y dedicación durante estos meses para ayudarme a realizar un buen proyecto, por estar disponibles siempre que lo he necesitado y por aportar su gran experiencia en el tema que trata este trabajo.

En segundo lugar, quería también agradecer a todas las personas que, sin ser de mi familia más cercana, me han ayudado y motivado durante este periodo. Ellos son todos mis amigos, tanto los amigos que he tenido la suerte de conocer durante estos años en Valencia como los amigos de mi pueblo natal. Quería darles las gracias por estar siempre que les he necesitado.

Y, por último, no podía olvidarme de mi familia, mis padres, mi hermana, tíos, primos y abuelos, que son los que siempre están tanto en lo bueno como en lo malo y son los que me han enseñado la importancia de creer en uno mismo y de resolver todo lo que te propongas con humildad y sencillez. Quería dedicarles este trabajo a todos ellos.

Abreviaturas

ML – *Machine Learning*

DL – *Deep Learning*

CV – *Computer Vision*

CNN – *Convolutional Neural Network*

CVAT – *Computer Vision Annotation Tool*

VIA – *VGG Image Annotator*

FCN – *Fully Connected Network*

IoU – *Intersection Over Union*

ASPP – *Atrous Spatial Pyramid Pooling*

Tabla de contenidos

Índice de Figuras

Índice de Tablas

1. Introducción

El campo de la visión por computador[1] se centra en la creación de sistemas que, de forma autónoma, sean capaces de imitar tareas propias de la percepción visual humana y, en algunos casos, incluso superarla. La evolución de este ámbito se acelera con la llegada de los avances en inteligencia artificial y *machine learning*[2], los cuales permitieron la creación de herramientas cada vez más complejas que muestran un gran potencial en diversos sectores del mercado.

Algunas de las aplicaciones más conocidas de la visión por computador son el control de calidad en procesos industriales, los sistemas de conducción autónoma o la identificación y reconocimiento de objetos, entre otras.

La automatización del agarre robótico (Figura 1) es otra de las aplicaciones de la visión por computador y uno de los campos de investigación más activos en robótica en la actualidad. Tradicionalmente, esta aplicación ha requerido del conocimiento y la intervención humana para crear analíticamente un algoritmo específico para esta tarea. Pero este planteamiento resulta complejo y lento, y tiene limitaciones significativas para una aplicación generalizada. Por ello, y gracias a los exitosos resultados que los métodos de aprendizaje automático han conseguido, los investigadores están explorando el uso de dichos modelos en aplicaciones robóticas como esta.

En este proyecto se propone una solución para una tarea más concreta dentro de este campo, que es la de detección automática del agarre robótico para el objeto más accesible entre varios superpuestos. Por lo tanto, el propósito es el estudio de la tarea y los fundamentos teóricos detrás de esta para su posterior análisis con el fin de crear un sistema autónomo capaz de realizar, primero la detección del objeto más accesible entre todos los posibles, y segundo la detección del mejor agarre robótico para dicho objeto.

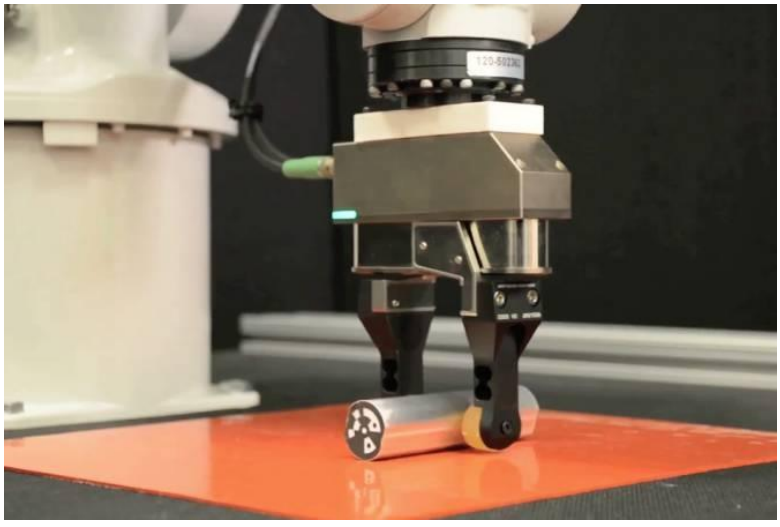


Figura 1. Agarre robótico en entorno industrial.

[1] https://es.wikipedia.org/wiki/Visión_artificial
[2] https://www.iberdrola.com/innovacion/machine-learning-aprendizaje-automatico

1.1. Motivación

El motivo de elección de este trabajo ha sido el hecho de pensar en la gran importancia que tiene actualmente el uso de inteligencia artificial y el aprendizaje automático profundo en entornos industriales, y el gran avance que estos pueden suponer en este ámbito. Hoy en día son cada vez más las empresas que están incorporando estas tecnologías con el objetivo de mejorar en precisión, tiempo y costo sus tareas. Mediante este proyecto se logrará automatizar la tarea de detección del agarre del objeto mejor posicionado para ser agarrado por un brazo robótico, el cual posteriormente y con el objeto agarrado, realizará la acción o movimiento que se le haya asignado (ya sea para una tarea de *picking* o transporte de materiales, por ejemplo).

1.2. Objetivos

El primer objetivo de este proyecto es conseguir un módulo basado en inteligencia artificial capaz de reconocer cuales de los distintos objetos (con la misma forma o estructura) de una imagen son los accesibles por el agarre del brazo robótico, es decir, aquellos objetos que no tienen algún tipo de impedimento u objeto que interfiera en su manipulación. Esto se va a conseguir mediante segmentación automática, que permitirá al sistema conocer la posición y estructura de los objetos mejor posicionados. Para ello, se necesita alcanzar otro de los objetivos del proyecto, conseguir un buen conjunto de datos con el que entrenar el modelo que realizará esta tarea.

Una vez detectados los objetos, el siguiente objetivo es extraer el mejor agarre para cada uno de ellos mediante el uso de técnicas de visión y las librerías `OpenCV`[3], `Numpy`[4] y `SciPy`[5] de Python. Este agarre vendrá descrito por el punto central del objeto (que en este caso coincidirá con el punto central del agarre), así como la anchura, el ángulo y la orientación del agarre.

[3] https://docs.opencv.org/master/
[4] https://numpy.org/doc/stable/
[5] https://docs.scipy.org/doc/scipy/

1.3. Estructura de la memoria

Respecto a la estructura sobre la que se apoya esta memoria, esta consta de los siguientes capítulos:

- **Capítulo 2. Estado del arte**: en este capítulo se expondrá una breve introducción acerca de las técnicas más utilizadas en tareas de reconocimiento y segmentación de objetos basadas en aprendizaje automático, así como en la detección automática de su agarre robótico, hasta el momento. También se comentarán algunas de las herramientas más utilizadas para el etiquetado de muestras como las extraídas en este proyecto.

- **Capítulo 3. Desarrollo del sistema de detección del objeto más accesible**: este capítulo va a mostrar el desarrollo paso a paso del sistema creado para la detección del objeto más accesible, con las fases llevadas a cabo para conseguirlo. Además, se expondrán las pruebas y comparaciones (de resultados de las métricas de evaluación) de las diferentes tecnologías y herramientas que se han realizado con el objetivo de conseguir el mejor modelo posible.

- **Capítulo 4. Sistema de extracción del mejor agarre**: en este se verá cómo, sobre la salida del sistema anterior, se puede extraer el mejor agarre del objeto más accesible mediante técnicas de visión por computador con la librería `OpenCV` de Python.

- **Capítulo 5. Conclusiones**: este contiene las conclusiones obtenidas del proyecto, así como la relación del contenido con los estudios cursados en el máster.

- **Capítulo 6. Trabajos futuros**: este último capítulo trata acerca de posibles trabajos futuros y extensiones a realizar sobre este proyecto.

2. Estado del arte

Antes de abordar el desarrollo del sistema de detección del mejor agarre, en este capítulo se hará una breve introducción acerca del contexto actual en el que se encuentra el campo de la visión por computador en relación con el reconocimiento de agarres y segmentación de objetos. Se expondrá una visión general de los distintos algoritmos, técnicas, herramientas y fundamentos teóricos utilizados actualmente para dicho propósito, y de esta forma, se comprenderá mejor el entorno en el que se encuentra este proyecto.

En primer lugar, se hará una breve introducción al *deep learning*, tras lo cual, se comentarán las técnicas y aplicaciones más recientes de DL en problemas de segmentación de objetos en imágenes. A continuación, se verán algunas de las herramientas actuales usadas para el etiquetado y obtención del conjunto de datos necesario para el entrenamiento del sistema. Y, por último, se presentarán algunas de las actuales soluciones en visión que se pueden encontrar para la extracción del mejor agarre posible de un determinado objeto.

2.1. *Deep Learning*

El *deep learning*[6] se conoce como un subconjunto del *machine learning* y de la inteligencia artificial que pretende imitar la forma en la que trabajan las conexiones neuronales biológicas del cerebro humano. Mientras que el *machine learning* trabaja con algoritmos de regresión o con árboles de decisión, el *deep learning* hace uso de las llamadas redes neuronales.

Una red neuronal consiste principalmente en unidades de proceso muy simples llamadas neuronas, las cuales se agrupan en capas y poseen un valor real por cada conexión con la capa anterior llamado peso. Los pesos se usan para procesar un valor de entrada. Dependiendo de las conexiones de cada capa, estas se pueden clasificar en tres tipos diferentes:

- Capas de entrada: en la mayoría de las topologías solo hay una capa de entrada, aunque en algunos diseños hay más. En las capas de entrada se inyectan los datos que representan el *input* de la red.

- Capas de salida: al igual que en el caso de las de entrada, en la mayoría de las topologías solo hay una capa de salida, pero en algunos diseños hay varias. Los valores de las capas de salida tras procesar una muestra son los que se utilizan, según la tarea, para tomar decisiones o proporcionar predicciones de valores.

- Capas ocultas: una o más capas ocultas conectan la(s) capa(s) de entrada con la(s) de salida. En la secuencia de capas ocultas se van obteniendo representaciones intermedias de los datos de entrada para detectar patrones relevantes para la tarea. Gradualmente, las primeras capas son sensibles a patrones pequeños, mientras que las neuronas de las ultimas capas ocultas (las más cercanas a la salida) se activan para patrones más complejos. Por ejemplo, las neuronas de las primeras capas son sensibles a bordes de objetos, mientras que las neuronas de las últimas capas lo serán a patrones mucho más complejos correspondientes a objetos o rostros de personas.

Para que la red neuronal devuelva los resultados deseados para una determinada entrada, es necesario que las neuronas ajusten los pesos adecuados para la tarea. Esto se va a conseguir mediante el entrenamiento, que puede ser de dos tipos: supervisado y no supervisado.

El aprendizaje supervisado es el más común de los dos, para el cual se posee un conjunto de datos que contará con la información que vamos a suministrar a la red (*input*) y la salida que esperamos obtener de esta (*label* o etiqueta, o clase). En este entrenamiento, la red recibe como entrada los datos (imágenes, secuencias…) y los procesa para obtener una salida, la cual se comparará con la salida que se espera del modelo. Esta comparación se realiza mediante las funciones de pérdida. Entonces, con la perdida obtenida en la capa de salida se actualizarán los pesos de la red (mediante el algoritmo de *backpropagation* y un optimizador), de manera que el resultado de procesar la entrada se acerque cada vez más al objetivo. Si el sistema está bien construido y es capaz de resolver el problema, podrá converger dando lugar a un modelo capaz de producir la salida esperada para la entrada del modelo.

[6] https://es.wikipedia.org/wiki/Aprendizaje_profundo

El aprendizaje no supervisado, por el contrario, se utiliza en conjuntos de datos donde no se dispone de la etiqueta o clase de cada muestra, o bien para hacer un preprocesado de los datos previo a una tarea de aprendizaje supervisado. Su objetivo principal es detectar patrones que se repiten con mayor frecuencia y que son distintos entre sí para separar y clasifiquar los datos en diferentes grupos. En el caso de las redes neuronales, suele usarse como una forma de inicializar los pesos antes de añadir la capa de salida y utilizar aprendizaje supervisado.

En este proyecto, requerimos del uso de redes neuronales (con aprendizaje supervisado) para el procesamiento de imágenes, y es por esto por lo que también se hablará en el estado del arte acerca las redes más eficientes actualmente en este ámbito, las redes neuronales convolucionales[7] (CNN).

2.1.1. Redes neuronales convolucionales

En este tipo de redes neuronales, las capas convolucionales procesan la imagen de entrada mediante una ventana con tamaño fijo (*kernel*), que va desplazándose hasta extraer valores para toda la imagen. Para extraer los valores, se multiplica el valor de los por los pesos del *kernel* sobre cada submatriz de la imagen del mismo tamaño del *kernel*. Es lo que se conoce como campo receptivo. Se puede ver un ejemplo del funcionamiento en la Figura 2.

El principal parámetro de configuración de las capas convolucionales es el número de filtros de salida, el cual determina la profundidad del tensor de salida. La estrategia para formar el tensor de salida consiste en que la capa convolucional calcula, por cada filtro, uno de los niveles de profundidad del tensor de salida. Esta estrategia permite a las capas convolucionales extraer varios tensores de profundidad 1 a partir del tensor de entrada.

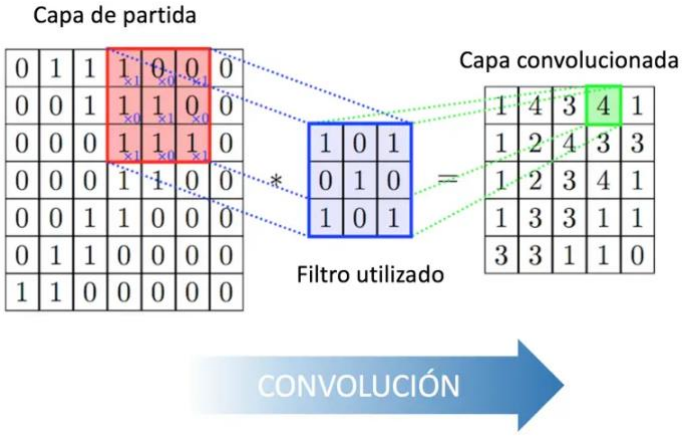


Figura 2. Ejemplo de convolución básica.

[7] https://es.wikipedia.org/wiki/Red_neuronal_convolucional

2.2. Segmentación de objetos

La segmentación [1] es un proceso de clasificación por píxeles. En esta se asignará una categoría a cada píxel de la imagen analizada, y tiene como principal objetivo localizar y diferenciar regiones con significado (ejemplo en la Figura 3).

La segmentación se usa tanto para localizar objetos como para encontrar sus bordes dentro de una imagen. El resultado de la segmentación de una imagen es un conjunto de segmentos que cubren toda la imagen sin superponerse. Esta se puede representar como una imagen de etiquetas (una etiqueta para cada píxel) o como un conjunto de contornos.

Figura 3. Ejemplo de segmentación.

2.2.1. Tipos de segmentación

Se pueden distinguir varios tipos de segmentación como se puede ver a continuación:

- Segmentación binaria: únicamente distingue entre dos categorías.
- Segmentación múltiple: distingue entre más de dos categorías.
- Segmentación asistida o semiautomática: se alcanzan mediante algoritmos que requieren anotaciones previas, señalando regiones a analizar o a distinguir. En algunos casos es posible automatizar al completo estas anotaciones haciendo uso de otros algoritmos de visión artificial, como en este proyecto, que haremos uso del DL para automatizar el proceso.
- Segmentación semántica: distingue categorías de alto nivel significante, normalmente objetos.
- Segmentación instanciada: distingue individuos de la misma categoría.
- Segmentación panóptica: combina la segmentación semántica e instanciada, reconociendo individuos de varias categorías.

En este proyecto, donde únicamente se desea diferenciar entre la categoría de objeto o fondo, la segmentación realizada es semántica, binaria y automática, ya que se pretende distinguir una categoría de alto nivel significante, que sería el objeto accesible, de la segunda categoría, que sería el fondo o resto de la imagen, mediante el uso de las redes neuronales convolucionales que se especifican en los próximos capítulos.

2.2.2. Segmentación basada en aprendizaje automático

Actualmente, existen varias alternativas de métodos de segmentación que no hacen uso del machine learning. Algunos de los métodos más destacables son la umbralización, el algoritmo *split and merge*, los algoritmos de rellenado o aumento de regiones (que se puede ver en [1]), el algoritmo *Watershed* y el algoritmo *GrabCut*. A continuación, se explicará brevemente en qué consisten algunos de los más destacables.

- La umbralización [1] es la segmentación más básica y simple, y es un ejemplo de segmentación binaria ya que trabaja sobre cada píxel de forma independiente, clasificándolos por su intensidad comparada con un umbral dado (si el píxel es más claro corresponde al frente, si no, al fondo).

- El algoritmo Watershed [2] (1979) requiere asignar etiquetas a algunos píxeles, a los cuales llamamos semillas. De este modo, el algoritmo ampliará paulatinamente la región a partir de ellas buscando bordes entre segmentos.

- El algoritmo GrabCut [3] (2004) es un ejemplo de segmentación binaria más reciente que tiene como objetivo capturar el objeto dominante de la imagen y, para ello, requiere que el usuario recuadre el objeto, tomando así la zona de fuera del recuadro como semilla del segmento del fondo.

- El algoritmo split and merge [4] que consiste en un proceso iterativo que realiza primero una división en regiones para, posteriormente, unir aquellas regiones vecinas similares. Este algoritmo finalizará cuando las segmentaciones sean uniformes y suficientemente pequeñas.

Cada uno de ellos, funcionará mejor o peor para un determinado problema y la única forma de conocer esto es validándolo de una manera eficaz y conveniente.

Pero, como en cualquier paso de la visión artificial, el *machine learning* puede ser utilizado como alternativa a estos métodos más clásicos en el contexto de la segmentación de imágenes. El DL se presenta como una solución mucho más óptima e inmediata para realizar esta tarea.

Cuando hablamos de segmentación de imágenes, los elementos clave en la arquitectura de la red neuronal necesaria son un codificador y un decodificador (arquitectura convolucional de codificador-decodificador). En el codificador, también conocido como *downsampling*, a partir de la imagen de entrada se generará un mapa de características de baja resolución. Tras este, en el decodificador, también conocido como *upsampling*, se aumentará la dimensión espacial del mapa de características anterior con el fin de obtener una imagen del mismo tamaño que la imagen de entrada. Finalmente, la salida final del decodificador se alimenta a un clasificador *softmax* multiclase, produciendo así las probabilidades de clase para cada píxel de forma independiente y extrayendo la segmentación.

Desde el desarrollo de las redes neuronales convolucionales (CNN), las técnicas de DL para la segmentación en visión por computador han avanzado a gran velocidad, proponiendo múltiples estrategias y modelos para resolver este problema. A continuación, se comenta cómo se han ido desarrollando y como han avanzado las técnicas de esta tarea a lo largo de los últimos años.

2.2.2.1. Fully Convolutional Networks for Semantic Segmentation

En 2014, Long, Shelhamer y Darrell publican la FCN (*Fully Convolutional Networks for Semantic Segmentation*) [5], una de las primeras estructuras que populariza el uso de redes convolucionales end-to-end para la segmentación semántica.

Esta FCN hace uso de redes pre-entrenadas y capas de deconvolución para el remuestreo, introduciendo conexiones puente entre en codificador y el decodificador para sumar mapas de características. En la FCN, después de terminar la convolución en una red pre-entrenada (como por ejemplo VGG), los mapas de características se amplían mediante las capas de deconvolución, que, en vez de usar interpolación bilineal, aprenden la interpolación. Las conexiones entre codificador y decodificador se usan para obtener mejores resultados, ya que las capas de deconvolución pueden producir segmentaciones erróneas debido a la pérdida de información que producen las capas del codificador.

En la actualidad, existen 3 versiones de esta arquitectura:

- FCN-32: produce directamente el mapa de segmentación a partir de la convolución número 7 de la Figura 4, aplicando una sola deconvolución o convolución traspuesta[8] (tiene el efecto contrario a una convolución normal, es decir, aumenta el mapa) con *stride* 32.

- FCN-16: suma el mapa resultante de aplicar una deconvolución con *stride* 2 al mapa resultante de la convolución número 7, y el mapa del *max-pooling* número 4 de la Figura 4 para producir el mapa de segmentación. Tras esto, se aplica una sola deconvolución o convolución traspuesta con *stride* 16 para obtener el mapa final de segmentación.

- FCN-8: suma el mapa resultante de aplicar una deconvolución con *stride* 2 al mapa resultante de la convolución número 7, y el mapa del *max-pooling* número 4 de la Figura 4. El resultado pasará por una deconvolución con *stride* 2 y es sumado al mapa obtenido del *max-pooling* número 3. Finalmente, se aplica otra convolución traspuesta con *stride* 8 para obtener el mapa final de segmentación.

En la Figura 4 se puede ver un esquema detallado de la estructura de esta arquitectura, con cada una de sus versiones en orden de arriba a abajo respectivamente.

[8] https://datascience.stackexchange.com/questions/6107/what-are-deconvolutional-layers

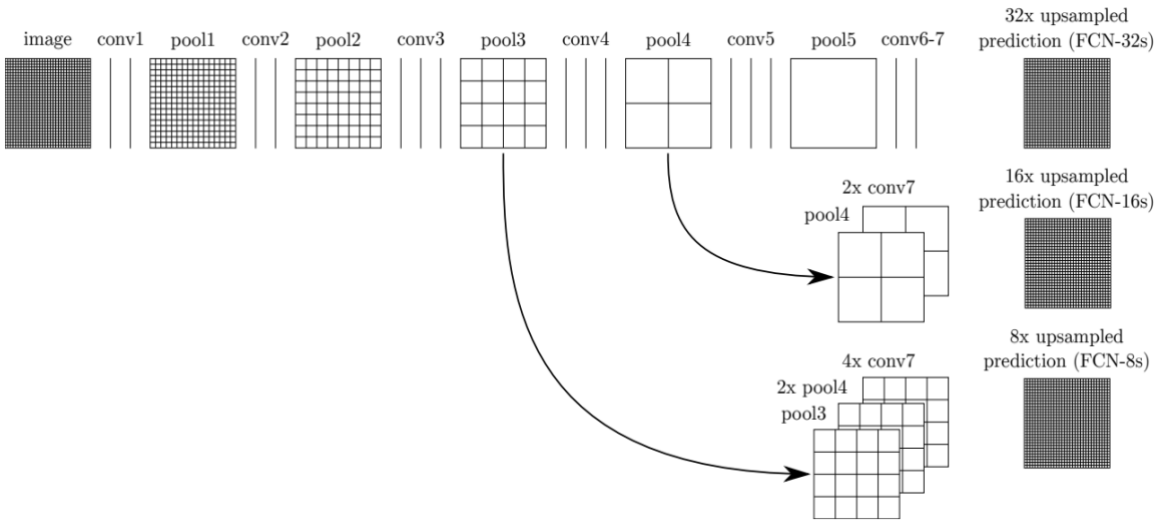


Figura 4. Arquitectura de las 3 versiones de la FCN para segmentación semántica.

2.2.2.2. Seg-net

En 2015 es publicada Seg-net [6] (*A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation*), cuya principal diferencia con la anterior, la FCN, es que Seg-net no cuenta con conexiones que sumen los mapas de características, sino que utiliza los índices obtenidos por las capas de *max-pooling* en las capas de decodificador. Seg-net es una arquitectura simétrica, es decir, cuenta con el mismo número de capas de codificador y decodificador como vemos en la Figura 5. En general y debido a esto, la Seg-net es más eficiente que FCN, pero sus resultados son peores.

En la Figura 5 se puede ver un esquema detallado de la estructura de esta arquitectura. Debido al uso de esta red neuronal en las pruebas de este proyecto, en el apartado 3.2.1.1 se puede ver una descripción más exhaustiva de esta arquitectura, donde se comentan algunos detalles sobre su implementación. En este capítulo simplemente se introduce comparándola con las demás arquitecturas que comparten su mismo propósito.

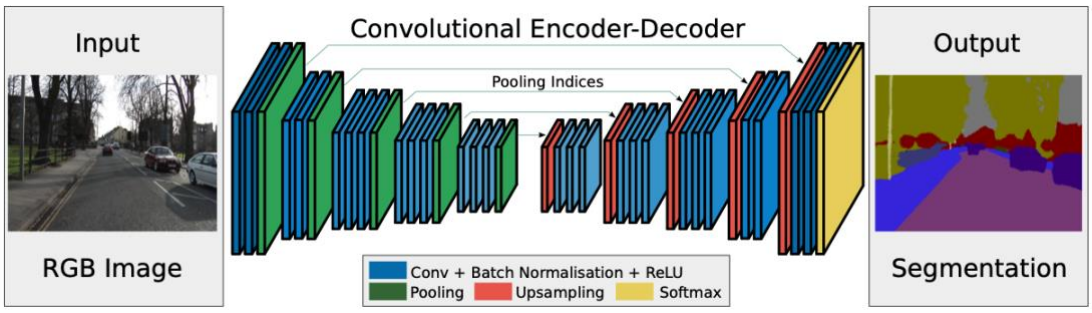


Figura 5. Arquitectura de la Seg-net.

A pesar de que Seg-net y FCN son 2 de las primeras arquitecturas codificador-decodificador, en el estado actual del arte Seg-net no es recomendado en la mayoría de los modelos de producción.

2.2.2.3. U-net

También en 2015, fue creada por Olaf Ronneberger, Philipp Fischer y Thomas Brox la llamada U-net [7], que se trata de una mejora y desarrollo de la FCN para segmentación semántica, y que permite trabajar con menos imágenes de entrenamiento produciendo segmentaciones más precisas.

U-net, en comparación con FCN, se trata de una arquitectura simétrica (al igual que Seg-net) ya que cuenta con las mismas capas de codificador y de decodificador. Por ello, U-net tiene una conexión por cada una de las capas de codificador o decodificador (en U-net las conexiones son llamadas *skip-connections* y unen las capas correspondientes al mismo nivel). Esto conlleva a que la red tendrá un mayor número de mapas de características en la fase de decodificación, lo que permite transferir más información.

En FCN, las conexiones únicamente suman el mapa de características de la capa de codificador al mapa de características correspondiente del decodificador, mientras que en U-net se concatenan. Los resultados de esta concatenación pasan por algunas capas de convolución para un procesamiento adicional.

En la Figura 6 se puede ver un esquema detallado de la estructura de esta arquitectura. Al igual que sucede con la Seg-net, debido al uso de esta red neuronal en las pruebas de este proyecto, en el apartado 3.2.1.2 se puede ver una descripción más exhaustiva de esta arquitectura, donde se comentan ciertos detalles sobre su implementación. En este capítulo simplemente se ha introducido comparándola con las demás arquitecturas con el mismo propósito.

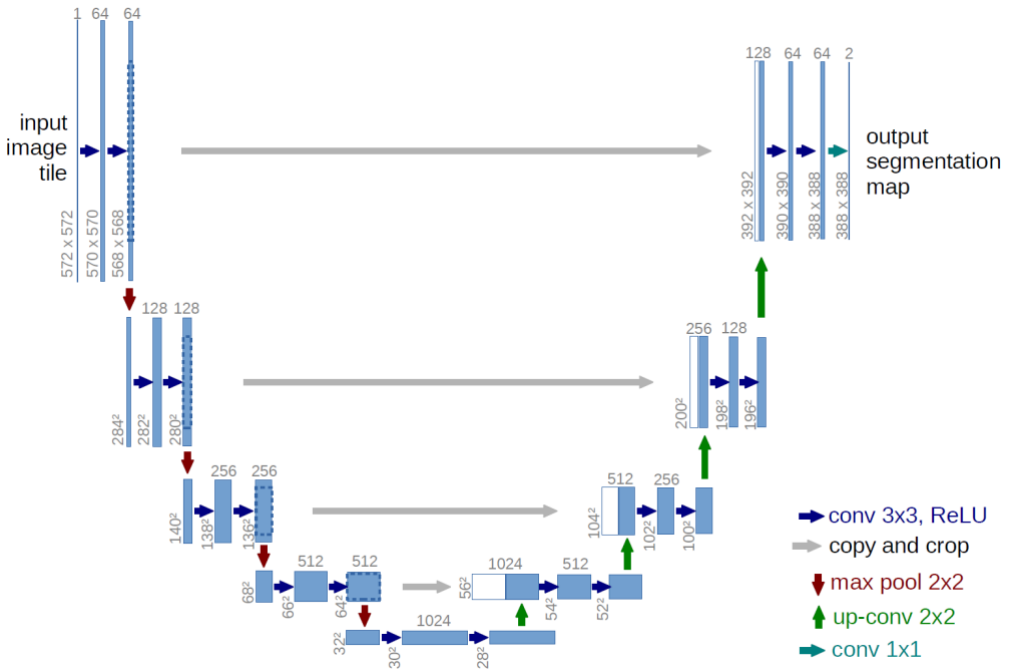


Figura 6. Arquitectura de la U-net.

2.2.2.4. DeepLab

DeepLab [8] es otra arquitectura de código abierto diseñada por Google para la segmentación semántica. Al igual que las anteriores, cuenta con una estructura de codificador-decodificador, pero cuenta con la diferencia de que usa convoluciones dilatadas (también conocidas como *atrous convolutions*[9]) para realizar el *up-sampling* en el decodificador.

Como se ha comentado en las anteriores arquitecturas, debido al uso repetido de *max-poolings* en las capas del codificador, la resolución de los mapas de características resultantes se reduce significativamente. Por ello, estas arquitecturas introducían capas de deconvolución para aumentar la muestra del mapa resultante. Sin embargo, esto requiere de memoria y tiempo adicionales, y por ello se introduce como alternativa la convolución dilatada. Esta convolución permite ampliar el campo de visión de sus filtros sin aumentar el número de parámetros ni el cómputo requerido, ya que introduce huecos o espacios de píxeles en los filtros distintos de 0. El parámetro llamado *rate* es el que especifica el tamaño de los espacios entre los pixeles del filtro. Se ven unos ejemplos en la Figura 7.

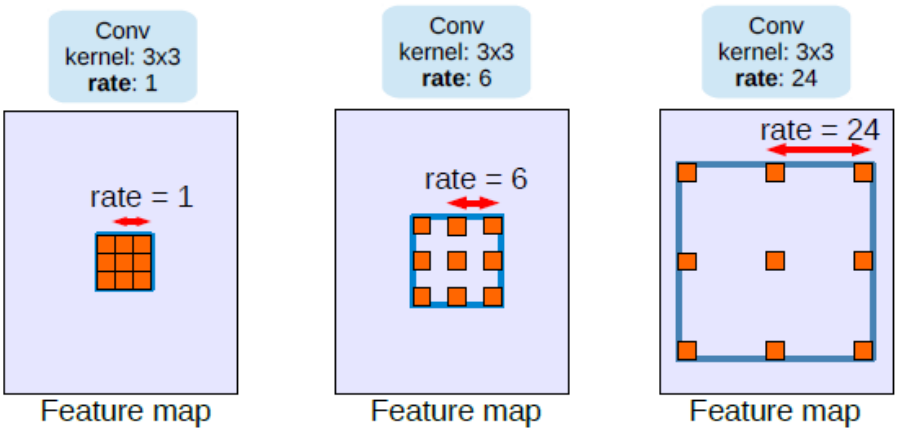


Figura 7. Ejemplos de *Atrous Convolution* o convolución dilatada.

Una vez explicado esto, se comentarán ahora las versiones de esta arquitectura que se han ido desarrollando a lo largo de los años.

- DeepLabv1 [8]: realiza varios avances sobre la FCN para segmentación semántica. Esta primera versión toma las imágenes como entrada y las pasa por las capas de convolución correspondientes al codificador, seguidas de una o dos capas de convolución dilatada. El mapa resultante se muestrea al tamaño original de la imagen mediante interpolación bilineal. El resultado de la segmentación final se ve mejorado gracias a que esta arquitectura aplica al final CRF (*conditional random field*), un modelo estocástico usado habitualmente para el etiquetado y segmentado de datos. CRF incorpora términos de suavizado que maximizan la posibilidad de que los pixeles similares tengan la misma etiqueta y pueden integrar relaciones contextuales entre clases de objetos.

[9] https://paragmali.me/atrous-convolutions/

Por lo tanto, esta versión resuelve o soluciona en parte el problema de anteriores arquitecturas que contaban con una resolución de características reducida y una precisión en la localización menor debido a la invarianza en las capas convolucionales del codificador. En la Figura 8 se puede ver un esquema detallado de su funcionamiento.

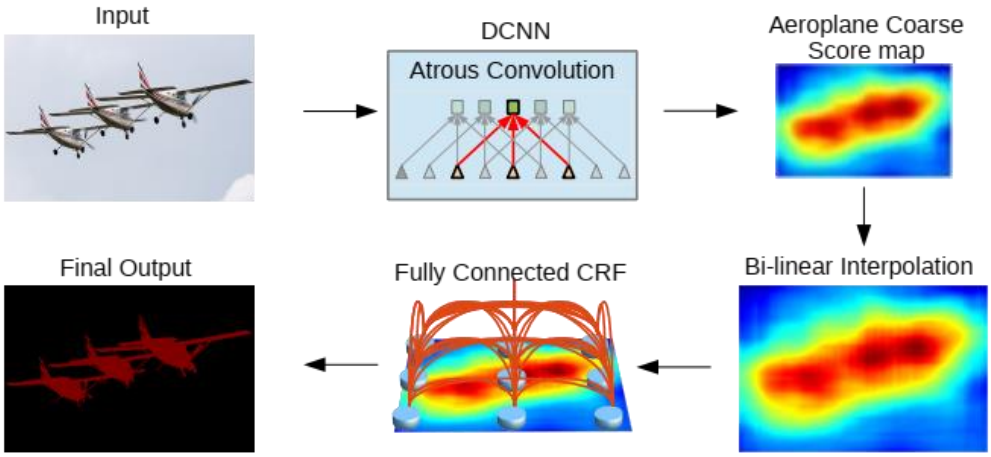


Figura 8. Fases de la arquitectura de DeepLabv1.

- DeepLabv2 [8]: esta segunda versión mejora la primera en la detección de objetos a múltiples escalas. Para ello, se introduce el uso de *atrous spatial pyramid pooling* (ASPP), que consiste en aplicar al mapa de entrada múltiples convoluciones dilatadas con diferentes *rates* para luego fusionarlos. Con esto, se tendrán en cuenta diferentes escalas de objetos, lo que ayudará a mejorar la precisión del modelo. El esquema del funcionamiento de ASPP se puede ver en la Figura 9.

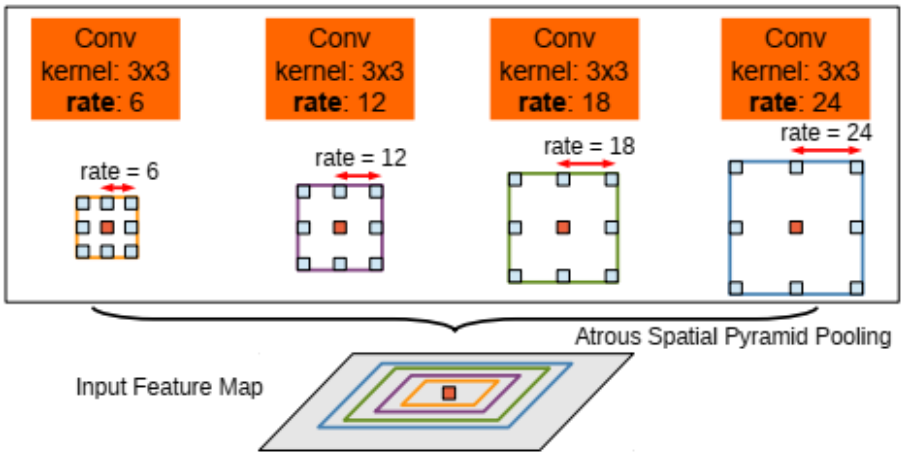


Figura 9. Arquitectura de DeepLabv2.

- DeepLabv3 [9]: la tercera versión adopta una arquitectura codificador-decodificador novedosa que incluye una nueva *depthwise separable convolution* o *atrous depthwise convolution* (Figura 10, apartado "c)"). Esta nueva convolución se consigue factorizando una convolución estándar en una *depthwise convolution* seguida de una *pointwise convolution* (que es una convolución 1x1).
  La *depthwise convolution* lleva a cabo una convolución independiente para cada canal de entrada (Figura 10, apartado "a)"), mientras la *pointwise convolution* (Figura 10, apartado "b)") combina la salida de la *depthwise convolution*.

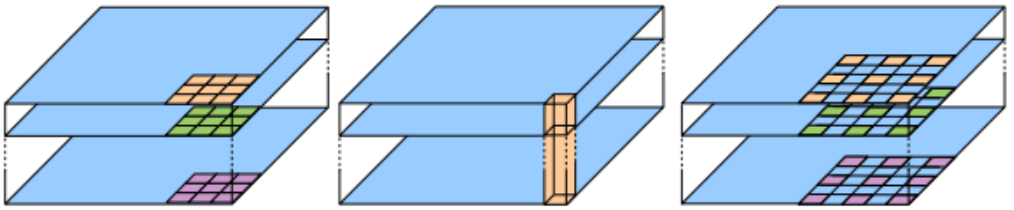


Figura 10. Comparación entre *depthwise, pointwise y atrous depthwise convolution*.

Con esto, el modelo consigue obtener de forma más nítida los límites de los objetos y además se logra un aumento de la eficiencia computacional.

- DeepLabv3+ [9]: mejora la arquitectura de DeepLabv3 añadiendo un módulo decodificador más efectivo para refinar aún más los resultados de la segmentación, haciendo hincapié en los límites de los objetos.
  En el codificador, todas las operaciones de *max-pooling* se sustituyen por *depthwise separable convolutions*. Y, en el nuevo decodificador, en lugar de usar un *up-sampling* bilineal con factor de 16, los mapas de características se muestrean con factor 4 para luego ser concatenados con sus correspondientes mapas de características de las capas de codificador con las mismas dimensiones. Tras la concatenación se aplican convoluciones 3x3 y *up-sampling* de factor 4. La salida de estas operaciones tendrá el mismo tamaño que la imagen de entrada.
  En la siguiente Figura 11 se puede ver un esquema más detallado del funcionamiento de esta última versión.

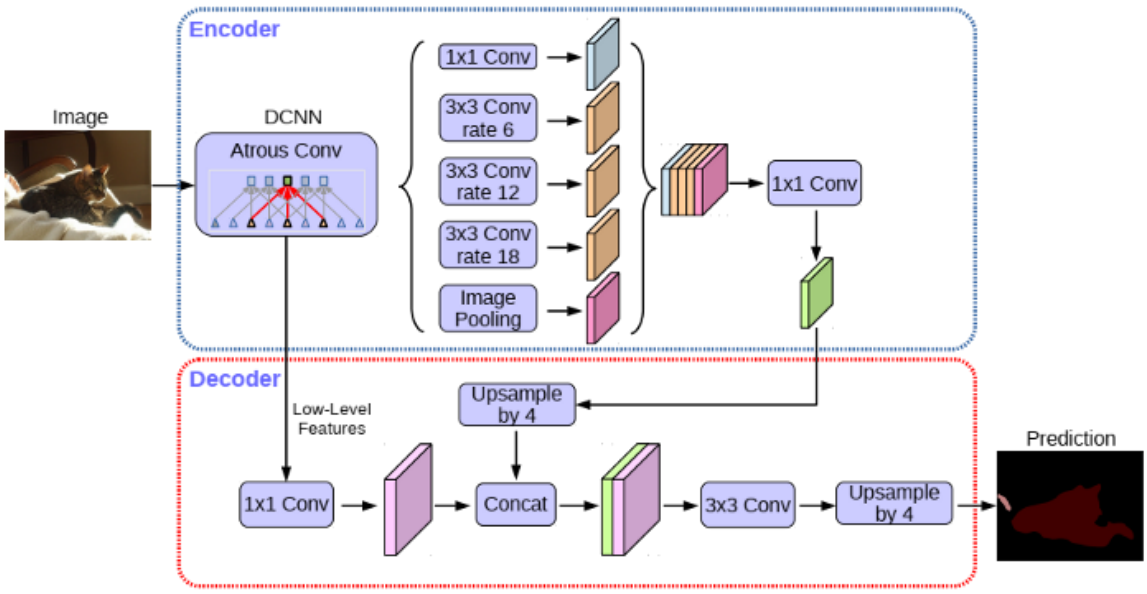


Figura 11. Arquitectura de DeepLabv3+.

2.3. Herramientas de etiquetado

Debido al gran y rápido avance del DL durante estos últimos años, han sido muchas las empresas o grupos de trabajo que han requerido del uso de herramientas y aplicaciones para llevar a cabo las tareas de etiquetado de imágenes o videos, con el objetivo de formar grandes conjuntos de datos correctos y precisos para el entrenamiento de sus modelos de *machine learning*.

Algunas de estas empresas hacen uso de sus propias herramientas, pero también hay muchas de ellas que optan por contratar los servicios de otras aplicaciones de etiquetado externas. Actualmente, las alternativas son inmensas, pero a continuación se mencionarán algunas de las más conocidas y usadas.

2.3.1. Labelme

Labelme[10] es una herramienta de etiquetado de código abierto escrita en Python y que utiliza la librería *Qt*[11] para su interfaz gráfica (Figura 12). Esta herramienta admite la anotación de polígonos (incluidos círculos, rectángulos, líneas, tiras de líneas y puntos) de forma manual en imágenes para la detección, clasificación y segmentación de objetos.

Labelme está inspirada en la herramienta de anotación del proyecto creado por el Laboratorio de Ciencias de la Computación e Inteligencia Artificial del Instituto Tecnológico de Massachusetts (CSAIL) llamado LabelMe[12]. Este proyecto se trata de un conjunto de datos dinámico, de uso gratuito y abierto a la contribución pública mediante su herramienta de etiquetado. Los usuarios realizarán cambios en las imágenes mediante esta hasta su correcto etiquetado, y estos se guardan y están disponibles abiertamente para que cualquiera pueda descargarlas desde el conjunto de datos de LabelMe. De esta forma, los datos estarán cambiando constantemente debido a las contribuciones de la comunidad de usuarios que utilizan la herramienta.

En Labelme solo es posible guardar los etiquetados como archivos JSON directamente desde la aplicación, pero en el repositorio de LabelMe se puede conseguir un script de Python para convertir estas anotaciones al formato PASCAL VOL. Otros formatos, como YOLO y COCO, no son compatibles en esta aplicación.

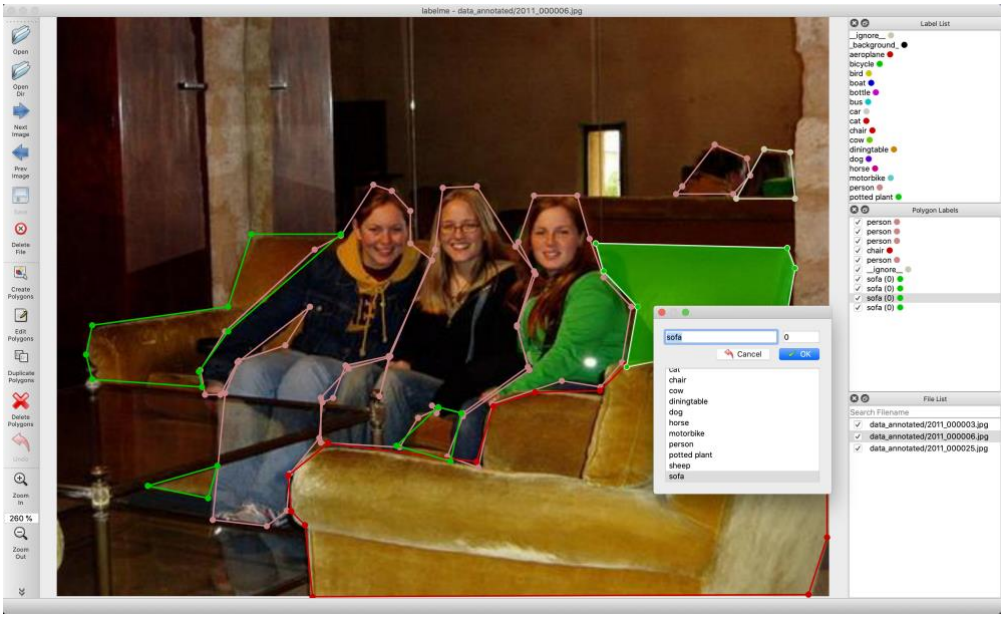


Figura 12. Interfaz de Labelme.

[10] https://github.com/wkentaro/labelme
[11] https://www.qt.io/qt-for-python
[12] http://labelme.csail.mit.edu

2.3.2. KeyLabs

KeyLabs[13] es otra plataforma (esta vez sin código abierto) de etiquetado de datos creada por expertos en anotación para ofrecer funciones de alto rendimiento y sistemas de gestión únicos. Esta aplicación permite asignar roles de usuario y permisos para cada proyecto individual. Los roles de usuario son: anotador, administrador de proyectos, superusuario y administrador. Podemos ver su interfaz en la Figura 13.

Esta herramienta cuenta con un historial de manejo de grandes conjuntos de datos manteniendo una buena eficiencia y precisión.

Algunas de sus características o funciones más significativas son:

- Permite a los anotadores definir la profundidad, o el orden z, de los objetos anotados.
- Los objetos podrán ser marcados cuando están fuera del cuadro de video durante un período de tiempo y luego capturarlos cuando reaparecen, utilizando las funciones de línea de tiempo de objetos.
- También puede otorgar a cada objeto una identidad visual única para ubicarlos en varios archivos multimedia.
- Admite la creación de metadatos, incluidas estructuras jerárquicas. Por ejemplo, los objetos se pueden vincular entre sí en una relación "padre/hijo".
- Es posible extraer las etiquetas en formato JSON.

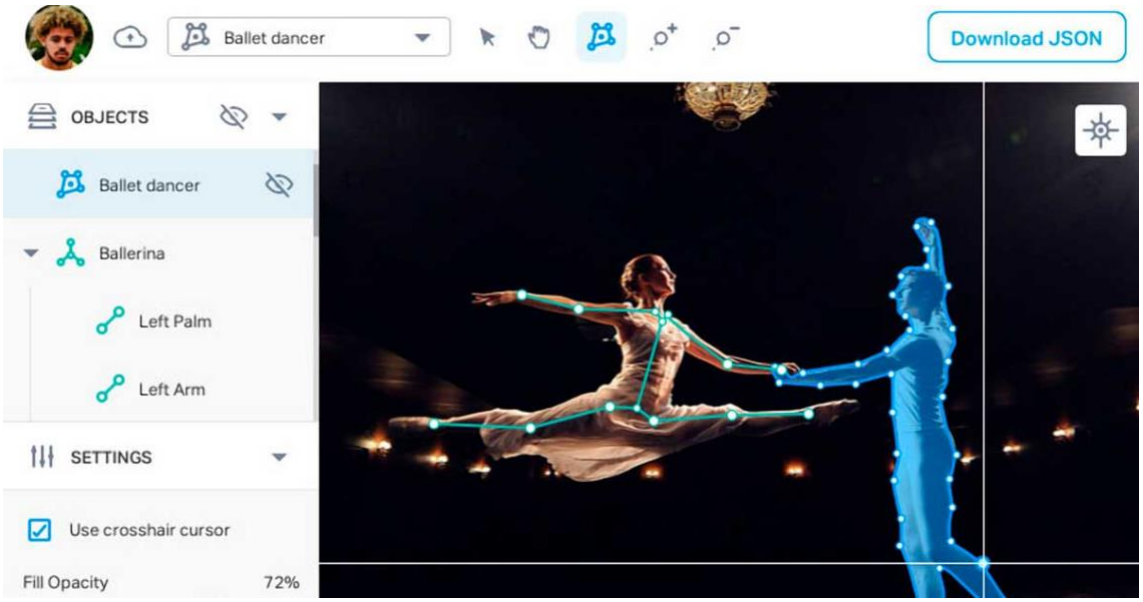


Figura 13. Interfaz de Keylabs.

[13] https://keylabs.ai/

2.3.3. CVAT

Computer Vision Annotation Tool[14] (CVAT) es una herramienta de anotación de imágenes y videos gratuita, de código abierto y desarrollada originalmente por Intel.

CVAT fue diseñado para ser usado por profesionales de etiquetado de datos, con una interfaz de usuario optimizada para tareas de anotación de visión artificial. Admite las principales tareas del aprendizaje automático supervisado: detección de objetos, clasificación de imágenes y segmentación de imágenes.

Además, tiene muchas características destacables, incluida la interpolación de cuadros delimitadores entre fotogramas clave, anotación automática mediante la API de TensorFlow OD y modelos de aprendizaje profundo en formato Intel OpenVINO[15] IR, accesos directos para la mayoría de las acciones críticas, tablero con una lista de tareas de anotación, LDAP[16] (para autorización) y autorizaciones básicas, etc.

CVAT está escrito principalmente en TypeScript, React, Ant Design, CSS, Python y Django. Se distribuye bajo la Licencia MIT [17] (del Instituto Tecnológico de Massachusetts) y su código fuente está disponible en GitHub.

Esta ha sido la herramienta elegida para el etiquetado de los datos de este proyecto debido a su fácil uso (además de suficiente para los objetivos requeridos) e interfaz sencilla e intuitiva (Figura 14).

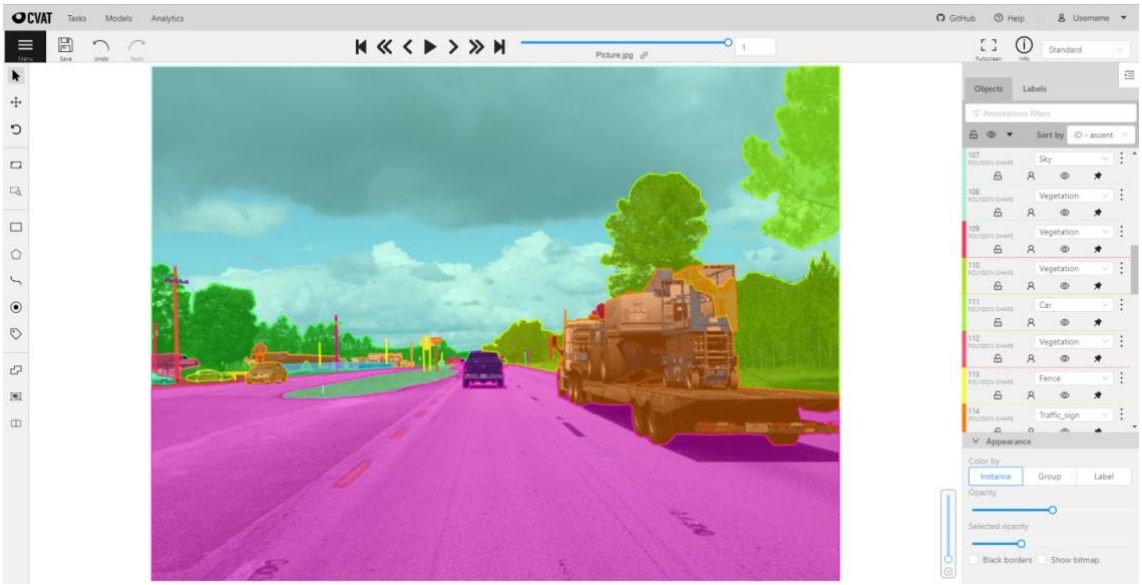


Figura 14. Interfaz de CVAT.

[14] https://cvat.org/
[15] https://docs.openvino.ai/latest/index.html
[16] https://www.techtarget.com/searchmobilecomputing/definition/LDAP
[17] https://es.wikipedia.org/wiki/Licencia_MIT

2.3.4. VGG Image Annotator

VGG Image Annotator[18] (VIA) es una herramienta de anotación manual simple e independiente para imágenes, video y audio. Este software (escrito en HTML, JavaScript y CSS) es liviano e independiente, y no requiere ninguna instalación o configuración, ya que se ejecuta únicamente en un navegador web (Figura 15).

VIA permite a los anotadores definir y describir regiones espaciales en imágenes o cuadros de video y segmentos temporales en audio o video. Estas anotaciones manuales se pueden exportar a formatos de datos de texto sin formato como JSON y CSV, de manera que se pueden procesar posteriormente con otras herramientas de software.

VIA también admite la anotación colaborativa de un gran conjunto de datos por parte de un grupo de anotadores. La licencia de código abierto BSD de este software permite su uso en cualquier proyecto académico o aplicación comercial.

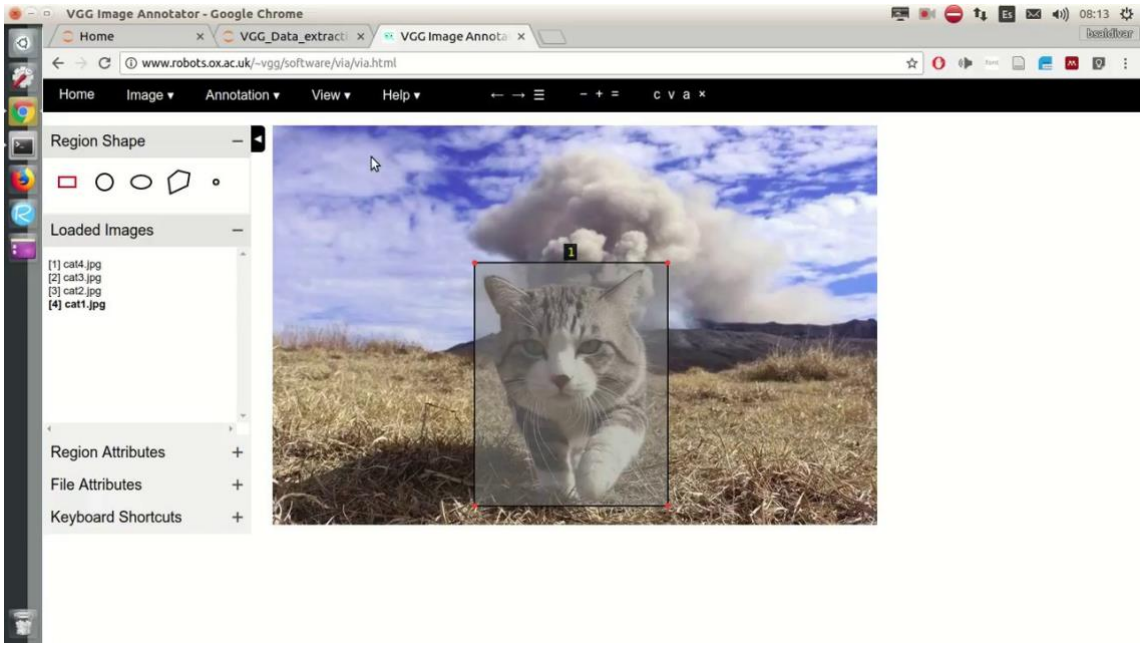


Figura 15. Interfaz de VGG Image Annotator.

[18] https://www.robots.ox.ac.uk/~vgg/software/via/

2.3.5. VoTT

Visual Object Tagging Tool[19] (VoTT) es una aplicación gratuita y de código abierto para la anotación y el etiquetado de imágenes desarrollada por Microsoft. El software fue desarrollado en el lenguaje de programación TypeScript y se utiliza para crear modelos de detección de objetos de extremo a extremo a partir de activos de imágenes y videos para algoritmos de visión por computadora.

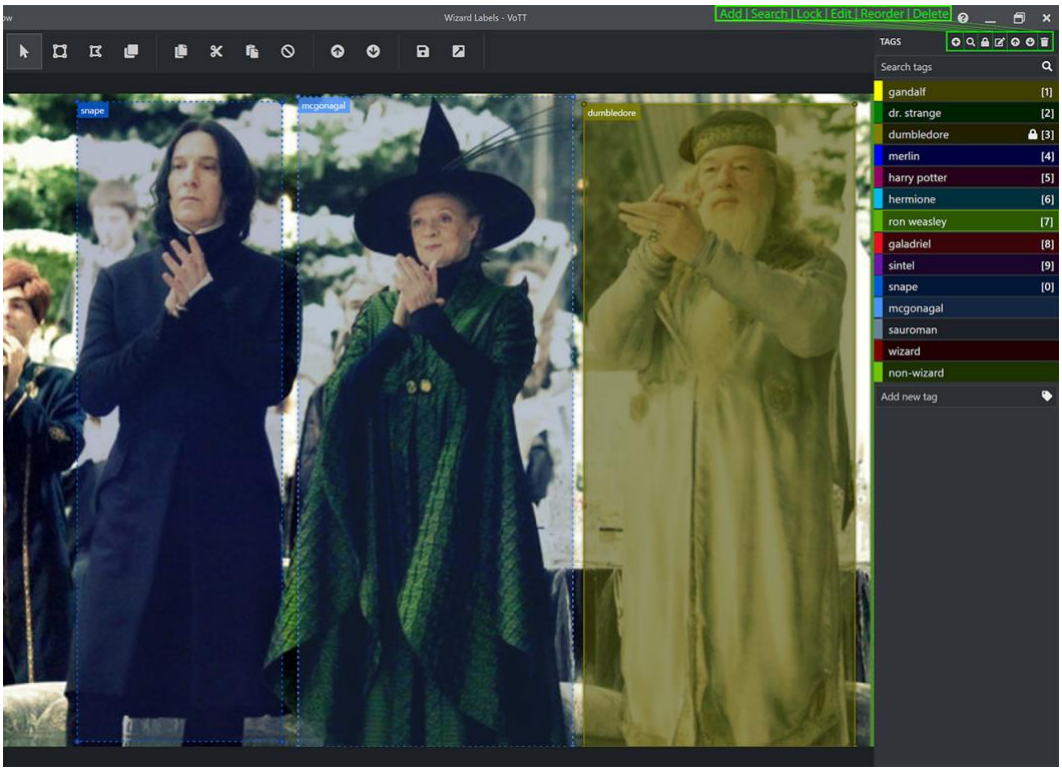


Figura 16. Interfaz de VoTT.

[19] https://github.com/microsoft/VoTT

2.4. Extracción de agarres

Y, para finalizar con el estado del arte, se expondrán en este apartado algunas de las soluciones que se pueden encontrar hoy en día para el problema de detección de agarres de objetos.

El proceso de la extracción del agarre de un objeto puede realizarse mediante varios mecanismos. En este proyecto se ha optado por realizarlo sobre la segmentación de un objeto, y trabajando mediante la librería OpenCV de Python con la estructura y la posición de este. Sin embargo, como en la mayoría de problemas en visión por computador, ya existen métodos de inteligencia artificial entrenados para realizar esta tarea de una forma más polivalente y eficaz cuando se conoce el objeto sobre el que hay que trabajar.

Antes de ver algunos de los modelos actuales, es importante hablar del conjunto de datos más destacable y usado para esta tarea, y sobre el que la mayoría de artículos y proyectos (como [10] y [11]) realizan sus pruebas con el objetivo de encontrar modelos que superen los resultados ya conseguidos. Este es *Cornell Grasp Dataset*[20], que cuenta con 885 imágenes de 240 objetos diferentes. Cada imagen tendrá varios rectángulos de agarre etiquetados como agarre válido o no válido para un brazo o mano robótica cuyo agarre se realiza mediante dos placas paralelas (como en la Figura 1). Estos rectángulos variarán en términos de orientación, localización y escala. En total hay 8019 agarres etiquetados, de los cuales 5110 son etiquetados como válidos y 2909 como no válidos. En la Figura 17 podemos ver algunos ejemplos de agarres en varios de los objetos del *dataset*.

Figura 17. Muestras de agarres en *Cornell Grasp Dataset*.

[20] http://pr.cs.cornell.edu/grasping/rect_data/data.php

Con esto, se comentará ahora cuáles han sido los tipos de redes neuronales que mejores resultados han obtenido en las investigaciones realizadas hasta el momento para este *dataset* en concreto.

Los primeros entrenamientos se realizaron con AlexNet [10,12], una arquitectura de red neuronal convolucional (CNN), diseñada por Alex Krizhevsky en colaboración con Ilya Sutskever y Geoffrey Hinton. AlexNet contiene ocho capas en total; las primeras cinco son capas convolucionales (las dos primeras seguidas por capas de *max-pooling*) y las últimas tres son capas completamente conectadas (*fully-connected*). Además, hace uso de la función de activación de ReLU, que muestra un rendimiento de entrenamiento mejorado sobre las funciones tanh y sigmoide. El resultado de *accuracy* obtenido con esta red fue de 88.0%.

En la Figura 18 podemos ver su arquitectura más detalladamente.

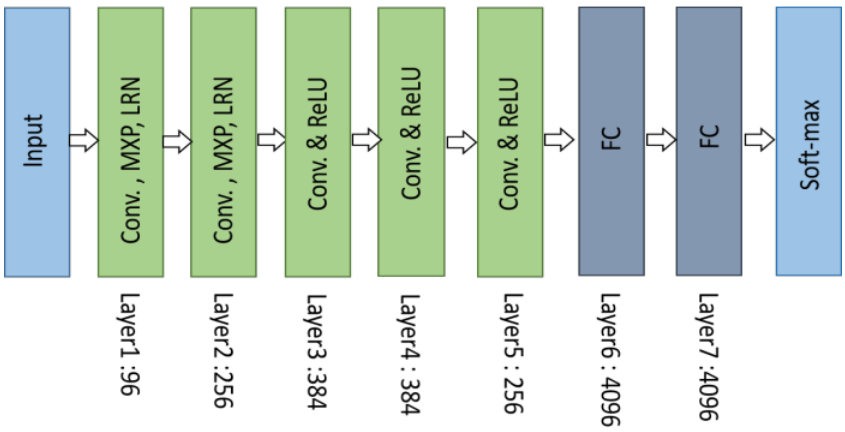


Figura 18. Arquitectura de AlexNet.

Un tiempo después de AlexNet, se comenzaron a realizar pruebas con la red residual ResNet-50 ([11,13]). Una red neuronal residual es una red que hace uso de atajos o conexiones de salto para moverse entre varias capas (por ejemplo, saltos de dos o tres capas con Batch Norm y no linealidad entre ellas), lo que significa que puede saltar el proceso de aprendizaje para algunas capas específicas. En la Figura 19 podemos ver más en profundidad el funcionamiento de estos saltos y su estructura. La idea de esta red y el salto entre capas es evitar que los gradientes se desvanezcan excesivamente al retropropagarse a capas anteriores repetidamente en el entrenamiento. Además, el salto hace la red más simple usando muy pocas capas durante la etapa de entrenamiento inicial, y acelera el aprendizaje considerablemente.

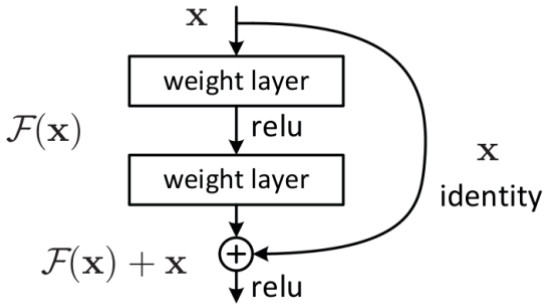


Figura 19. Estructura de una red neuronal residual.

La ResNet-50 es una de las redes residuales más conocidas y como su propio nombre indica cuenta con 50 capas de profundidad: 48 de ellas convolucionales, un Max-Pooling y un Average-Pooling. Su estructura detallada la vemos en la siguiente Figura (20).

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figura 20. Esquema de ResNet-50.

Los resultados obtenidos con esta red neuronal fueron levemente superiores a los obtenidos con AlexNet. Como se aprecia en [11], las 3 pruebas con ResNet-50 dieron los siguientes resultados de *accuracy*: 84.76% para ResNet-50 con SVM, 88.84% para ResNet-50 con activaciones ReLu y 88.53% para ResNet-50 con funciones de activación tanh y ReLu.

3. Sistema de detección del objeto más accesible

Una vez comentado el contexto actual en la que se encuentra el tema de este proyecto y vistas las posibilidades tecnológicas que existen a día de hoy para llevarlo a cabo, se pasará a ver en los siguientes apartados el sistema y la metodología por la que se ha optado en este caso para cada una de las fases del sistema.

Las fases en las que se ha dividido el sistema son: la creación del *dataset*, el entrenamiento de modelos y la evaluación de los mismos.

3.1. Creación del *dataset*

Para este proyecto se ha creado un conjunto de datos completamente nuevo con el que entrenar el modelo de detección del objeto más accesible, el cual cuenta con 250 muestras que se componen de:

- Una imagen RGB (3 canales) de dimensión 640x480 de la disposición de los objetos (en este proyecto, se ha trabajado con tuberías de pequeño tamaño como se puede ver en la Figura 21).

- Una imagen Depth de un solo canal que muestra la profundidad de cada píxel de la imagen RGB (y por lo tanto tiene la misma dimensión, 640x480).

- La segmentación de los objetos más accesibles correspondiente a la imagen (cada píxel se corresponde con 1 o 0 si el píxel pertenece a un objeto accesible o no, respectivamente).

En la Figura 21 se aprecia un ejemplo de una de las muestras del *dataset*.

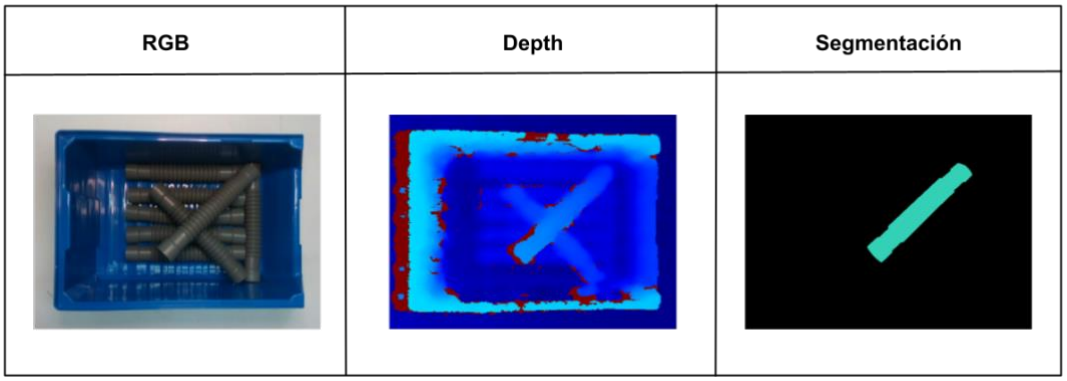


Figura 21. Muestra del conjunto de datos.

3.1.1. Extracción de imágenes RGB y Depth

La extracción de las imágenes RGB y Depth se ha conseguido mediante la cámara Intel RealSense D435[21] (Figura 22), que ofrece profundidad de calidad para una gran variedad de aplicaciones y, debido a su amplio campo de visión, es perfecta para aplicaciones como la robótica o la realidad virtual y aumentada, donde apreciar la mayor parte posible de la escena es de vital importancia. Cuenta con un alcance de hasta 10 metros y con un muy buen rendimiento también en su uso para distancias cortas.

Intel Realsense cuenta con una librería de Python llamada `pyrealsense2`[22], la cual junto con OpenCV ha permitido la extracción de las imágenes. `Pyrealsense2` cuenta con las funciones necesarias para detectar la cámara que se quiere usar, cambiar y establecer los formatos de imagen y profundidad, así como los tamaños de estas, alinear los *frames* en ambos formatos, establecer una escala de profundidad... entre otras configuraciones de la cámara.

[21] https://www.intelrealsense.com/depth-camera-d435/
[22] https://pypi.org/project/pyrealsense2/

Figura 22. Cámara Intel RealSense D435 y su aplicación.

A pesar de que Intel Realsense cuenta con una aplicación con la que se podría interactuar (Figura 22) y realizar todas estas funcionalidades, para este proyecto se ha decido usar el código Python. Este código funciona de la siguiente manera:

- Al ser ejecutado, el código se encarga de poner en marcha la cámara conectada y arrancar un directo tanto para RGB como para Depth con las funciones `config.enable_stream()` y `config.enable_stream()`, estableciendo también el tamaño y formato de ambas capturas.

- Con el directo puesto en marcha, si se presiona la tecla ESPACIO (función `is_pressed('space')` de la librería `keyboard`), se activará otro método que guardará los *frames* de ese instante tanto de RGB como de Depth.

- Este método, llamado `save_frame`, hace uso de las funciones `get_depth_frame()` y `get_color_frame()` de `pyrealsense2` para obtener los *frames* del instante en que se presiona la tecla. Este junto con `Numpy` para el manejo de matrices y `OpenCV` para escribir las imágenes, permitirá guardar los *frames* en nuestro conjunto de datos.

3.1.2. Segmentaciones (etiquetas)

En el caso del etiquetado (segmentado), este se ha conseguido con CVAT, la herramienta de etiquetado de imágenes y videos de visión por computador que se ha comentado en el capítulo 2 de la memoria. Su facilidad de uso y correspondencia con las exigencias del etiquetado del proyecto han sido las causantes de su elección.

En esta herramienta, ha bastado con crear un nuevo proyecto donde subir las imágenes RGB y establecer la configuración de las segmentaciones para poder comenzar a etiquetar. La configuración en este caso ha sido muy simple, ya que se ha establecido un solo tipo de etiqueta que se corresponde con la de objeto accesible, el resto de la imagen sería fondo. De esta forma, imagen a imagen se crearán tantas *shapes* o polígonos como objetos accesibles haya, y será el usuario el que deba ir dando a este polígono la forma del objeto deseado.

En la siguiente Figura (23) se podrá ver detalladamente cada una de las fases que se han realizado para conseguir el segmentado.

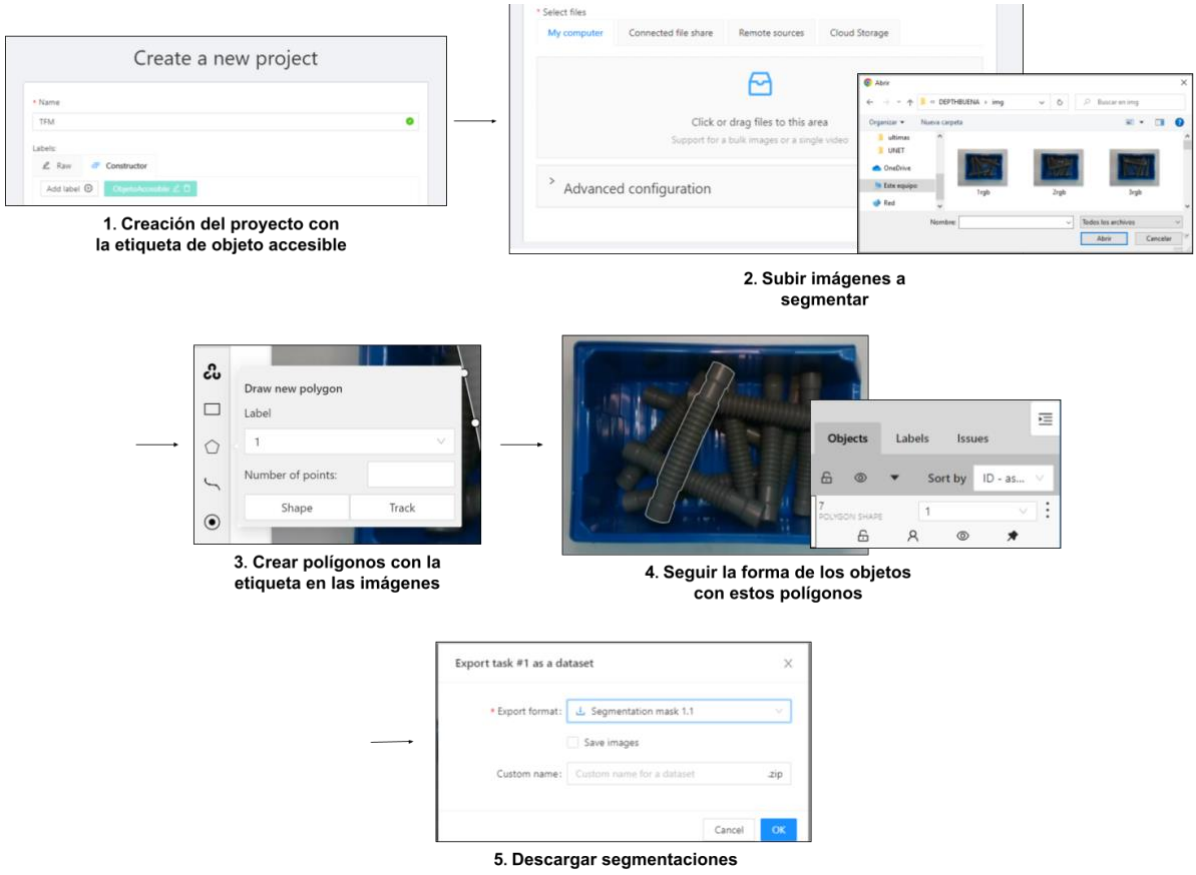


Figura 23. Fases del proceso de etiquetado con CVAT.

Con el conjunto de datos creado, en los siguientes apartados del capítulo se explicarán los diferentes tipos y estructuras de redes neuronales convolucionales que se usarán en el sistema de detección del objeto más accesible, las cuales se entrenarán y probarán con los datos obtenidos con el objetivo de compararlas y obtener el mejor modelo posible.

3.2. Entrenamiento de modelos

Tras la recopilación de los datos, se pasa a la fase de entrenamiento. El objetivo de este proyecto consiste en encontrar el mejor modelo posible, por lo que han sido muchas las pruebas y entrenamientos llevados a cabo.

En estas pruebas se han testeado diversas variaciones de arquitecturas de redes neuronales convolucionales y dentro de estas, diferentes valores de sus propios parámetros. A continuación, y antes de pasar con la fase de evaluación y resultados, vamos a explicar cada parámetro y arquitectura usada en los entrenamientos.

Destacar que toda la implementación de las arquitecturas se ha realizado con la librería de Python `TensorFlow`[23], una librería de código abierto para *machine learning* e inteligencia artificial creada por Google, que se especializa en el uso de redes neuronales profundas.

3.2.1. Arquitecturas

Han sido 3 las arquitecturas que se han entrenado en este proyecto. A continuación, se verán las características y diferencias que existen entre ellas, así como los métodos y funciones de `TensorFlow` que se requieren para su implementación.

3.2.1.1. Seg-net

Una Seg-Net [6] es una red neuronal totalmente convolucional que permite la segmentación de imágenes y que utiliza, como se ha comentado en el apartado 2.2.2.2 del estado del arte, un tipo de arquitectura codificador-decodificador con una capa final de clasificación de píxeles (Figura 5).

En la Seg-Net, cada capa de la red de codificadores realiza una serie de convoluciones (cada convolución se realiza mediante la función `Conv2D()`) que produce un conjunto de mapas de características. Tras las convoluciones, los mapas se normalizan con la función `BatchNormalization()` y se aplica una activación ReLU (`ReLU()`). A continuación, se realiza un *max-pooling* con una ventana de 2×2 usando la función `MaxPool2D()`, el cual decrementa el tamaño del mapa de características a la mitad del de su entrada. Tras este *max-pooling*, se pasa a la siguiente capa de codificador que sigue la misma estructura. Esto se repetirá hasta el comienzo del decodificador.

[23] https://www.tensorflow.org

El decodificador, en cambio, aumenta la muestra de los mapas de características obtenidos utilizando los índices de los *max-pooling* de la capa de codificador correspondiente. Estos mapas de características se pasarán por convoluciones para producir mapas de características densos, y a continuación, se les aplicará la normalización por *batches* (las funciones son las mismas que las usadas en la parte del codificador). El aumento de las muestras de los mapas se realizará mediante la función `UpSampling2D()`.

Cabe destacar que la capa de decodificador correspondiente a la primera capa de codificador produce un mapa de características multicanal, aunque la entrada del codificador tenga 3 canales (RGB). Esto es diferente a las otras capas de decodificador de la red, que producen mapas de características con el mismo número de canales que las entradas de su capa de codificador correspondiente.

3.2.1.2. U-net

Al igual que la Seg-Net, la U-net [7] es una red neuronal totalmente convolucional que permite la segmentación de imágenes y que utiliza un tipo de arquitectura codificador-decodificador como se ha comentado en el apartado 2.2.2.3 del estado del arte.

La diferencia con la Seg-Net es que la U-Net no reutiliza los índices de *max-pooling* en el aumento de mapas, sino que mediante el uso de las *skip-connections* correspondientes, concatena (mediante la función `Concatenate()`) el mapa resultante del *up-sampling* anterior con el mapa extraído tras las convoluciones de la capa de codificador correspondiente, aumentando así el tamaño de los mapas de características del decodificador como vemos en la Figura 6. En U-net, cada capa del codificador cuenta con 2 convoluciones de 3x3 seguidas de una activación ReLu, junto con el *max-pooling* de 2x2. En el decodificador, por su parte, cada capa cuenta con una deconvolución de 2x2, seguida de dos convoluciones de 3x3 y activación ReLu. La última convolución de 1x1 de la Figura 6 permite mapear el vector de características final al número de clases requerido para la tarea (en nuestro caso 2).

El resto de las funciones de `TensorFlow` usadas en la U-net (activación ReLu, convoluciones, *up-samplings*, *max-poolings*...) se corresponden con las comentadas en el apartado anterior (Seg-net).

3.2.1.3. Res-Unet

Una Res-Unet [14] es una *Deep Residual U-net*, es decir, una red neuronal residual con arquitectura codificador-decodificador. Esta red fue desarrollada por Zhengxin Zhang et al. para la segmentación semántica e inicialmente se utilizó para la extracción de carreteras a partir de imágenes aéreas de alta resolución en el campo del análisis de imágenes de teledetección. Más tarde los investigadores lo adoptaron para muchas otras aplicaciones, entre las que se destacan la segmentación de pólipos o de tumores cerebrales, entre otras.

La Res-Unet también es una red neuronal totalmente convolucional, pero está diseñada para obtener un alto rendimiento con menos parámetros. Mejora la arquitectura U-net existente aprovechando tanto su arquitectura como el *Deep Residual Learning*.

En lugar de las dos convoluciones de 3x3 seguidas de una función de activación ReLU (de la U-net), en la Res-Unet, estas capas se sustituyen por un bloque residual como los vistos en la Figura 24.

El uso de bloques residuales (comentados en el apartado 2.4 de la memoria) permite construir una red más profunda sin el problema de desvanecimiento de gradientes y ayuda al fácil entrenamiento de la red gracias al uso de las conexiones de salto para moverse entre capas.

La función `Add()` tras cada bloque es la que permite conseguir el efecto de red residual, que seguiría su ecuación:

$$F(x) + x$$

de la Figura 19. En la Figura 24 se puede ver la estructura de esta red al completo.

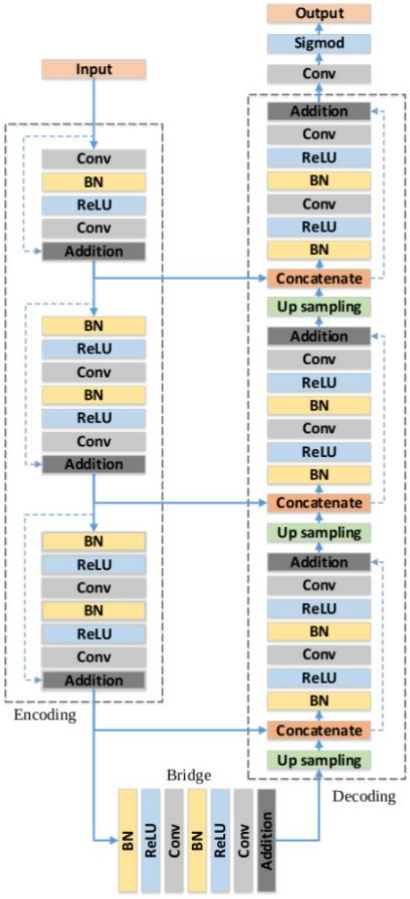


Figura 24. Arquitectura de Res-Unet.

3.2.2. Parámetros y modificaciones

Es importante destacar que las arquitecturas de las redes neuronales convolucionales comentadas en el apartado anterior son las arquitecturas estándar que se pueden encontrar en sus respectivas documentaciones. Sin embargo, para la realización de las pruebas de este proyecto y con el objetivo de encontrar el mejor modelo posible, estas arquitecturas se han modificado en algunos casos.

Los parámetros que se varían junto a los tipos de redes en cada una de las pruebas realizadas se comentan y explican a continuación.

3.2.2.1. Entrada de la red (inclusión del canal de profundidad)

En las arquitecturas usadas para las pruebas de este sistema, normalmente, la entrada consiste en una imagen RGB de tres canales. Sin embargo, para este proyecto se ha querido probar a incluir un cuarto canal en el *input*, que contendrá la información de la profundidad de la imagen (imagen Depth en el conjunto de datos). De esta forma, la red estará aprendiendo sobre más información y se pretende comprobar si debido a ello se pueden crear modelos más precisos.

Para que el uso de la profundidad funcione correctamente, es necesario que los valores del canal de esta queden en la misma escala a los de los canales RGB. Por lo tanto, se ha optado por pasar los valores de profundidad de cada píxel a una escala entre 0 y 255 con la siguiente formula.

$$\frac{255 * Depth}{\max(Depth)}$$

Una vez se tienen todos los canales escalados correctamente, se normalizarán a valores entre 0 y 1 (dividiendo por 255). Esto es necesario para pasar las muestras como entrada a la red.

En la Figura 25 se puede ver como quedaría la estructura de ambas entradas.

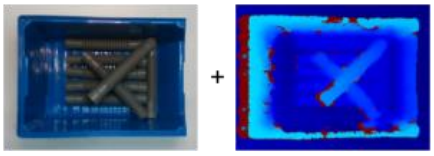
RGB	RGBD
[[230 100 104] [190 130 215] . . . [38 121 19]] / 255.0	[[230 100 104 12] [190 130 215 100] . . . [38 121 19 47]] / 255.0
	

Figura 25. Entrada de la red con y sin canal de profundidad.

3.2.2.2. Epochs y Batch Size

Todas las muestras de un conjunto de entrenamiento se utilizan repetidamente para entrenar una red neuronal. Se define *epoch* [15] como una iteración en la que se utilizan todas las muestras del conjunto una vez para ajustar los pesos de la red neuronal. Para que el ajuste de pesos sea más efectivo, las muestras de entrenamiento no se utilizan todas al mismo tiempo, sino que se agrupan en lo que se conoce como *batches*. Se define el *batch size* como el número de muestras que se utiliza al mismo tiempo para ajustar los pesos de la red. De esta manera, dentro de una *epoch* se itera para ir procesando las muestras *batch* a *batch*. Es habitual que las muestras de un batch se seleccionen aleatoriamente, tal que, en cada *epoch* las muestras se procesan en un orden distinto.

Para todas las pruebas de este proyecto se han realizado 10 *stages* de 100 *epochs* cada una, es decir, se han realizado un total de 1000 epochs por entrenamiento. La división de las *epochs* en *stages* se debe a que en cada una de ellas se ha realizado una evaluación con el objetivo de comprobar cómo evolucionan las segmentaciones predichas.

En cuanto al *batch size*, la mayoría de las pruebas se han realizado con un tamaño de *batch* de 32 muestras, pero se ha probado también con 16.

3.2.2.3. Optimizador

El optimizador[24] en una red neuronal es el encargado de generar pesos cada vez mejores, calculando el gradiente de la función de coste (derivada parcial) por cada peso de la red. Para minimizar el error, los pesos se modificarán en la dirección negativa del gradiente. Y de cara a agilizar la convergencia de la función de coste hacia su mínimo, el vector de gradiente se multiplica por la tasa de aprendizaje o *learning rate* (lr). El conjunto de métodos iterativos de reducción de la función de error (en búsqueda de un mínimo local) serán los métodos de optimización basados en el descenso de gradiente. Para este proyecto han sido dos los optimizadores que se han probado, Adam y SGD (Stochastic Gradient Descent).

El optimizador SGD [16] es el método básico de optimización de redes neuronales, el cual sigue la siguiente ecuación para su actualización de pesos:

$$W = W - lr * \Delta W$$

siendo W la matriz de pesos y ΔW su derivada. Aunque en este proyecto se ha hecho uso únicamente de esta versión de SGD, con *learning rate* fijo, existe una variante de este que incluye el llamado *momentum*. El *momentum* o inercia permite acelerar el descenso de gradiente en direcciones similares a las anteriores. Esto se realizará mediante la comparación del vector que representa la media de los anteriores vectores de descenso y el nuevo vector. Si el nuevo vector es similar se acelera su descenso.

Aunque SGD es usado como método de optimización en muchos problemas de ML, también se pueden encontrar los llamados métodos adaptativos, que en lugar de tener una misma tasa de aprendizaje global para todos los pesos (como en SGD), adapta un *learning rate* para cada peso individualmente. Adam será un método adaptativo.

[24] https://velascoluis.medium.com/optimizadores-en-redes-neuronales-profundas-un-enfoque-práctico-819b39a3eb5

Adam [17] combina las ventajas de los optimizadores AdaGrad y RMSProp. AdaGrad en vez de considerar un valor uniforme para todos los pesos, mantiene un *learning rate* específico para cada uno de ellos. Partiendo del *learning rate* inicial, AdaGrad lo escala y adapta para cada dimensión con respecto al gradiente acumulado en cada iteración. Y RMSProp por su parte también mantiene una tasa de aprendizaje diferente para cada dimensión, pero en este caso su escalado se realiza dividiéndolo por la media del declive exponencial del cuadrado de los gradientes, es decir, en función de lo rápido que estos están cambiando. Por lo tanto, Adam mantiene un factor de entrenamiento por parámetro y además de calcular RMSProp, cada factor de entrenamiento también se ve afectado por la media del momentum (que acelera el descenso en direcciones similares a las anteriores) del gradiente. En todas las pruebas con el optimizador Adam, el *learning rate* inicial ha sido de 0.00001.

3.2.2.4. Profundidad del Codificador/Decodificador

Como hemos visto en las arquitecturas de los apartados anteriores del capítulo, en los codificadores y decodificadores vemos un número concreto de *max-poolings* y *up-samplings* respectivamente, que es 4 (lo que supone 5 capas de codificador y de decodificador). Este es el número que aumentaremos o disminuiremos en las pruebas, de manera que se añadirán o quitarán capas de codificación y decodificación (quedando siempre el mismo número en ambos) para entrenar nuevos modelos y encontrar con qué profundidad de red encontramos el mejor resultado.

Es importante destacar que, en todas las pruebas (y todas las arquitecturas), en la primera capa de codificador las convoluciones cuentan con 32 filtros. A partir de esta, los max-poolings, al ser de 2x2 disminuirán el tamaño de los mapas a la mitad por cada capa, y el número de filtros de las convoluciones de las capas se duplica (es decir, en orden de capas sus respectivas convoluciones tendrán 32,64,128,256,512,1024,2048... filtros).

3.2.2.5. Data Augmentation

Aunque en todos los entrenamientos se ha contado con el mismo Data Augmentation sin variaciones, se comentarán las características de este. El Data Augmentation se ha llevado a cabo en la fase de carga de los datos (fichero *data_loader.py*) para el entrenamiento, mediante el uso de funciones de OpenCV. Este realiza los siguientes cambios en las imágenes para aumentar los datos:

- Una rotación de la imagen 180 grados mediante la función `cv2.rotate()`.
- Un factor de *zoom* de 0.7 y 1.3, lo cual se podrá aplicar con la función `cv2.resize()`.
- Un *flip* sobre el eje vertical mediante `cv2.flip()`.

Destacar que la rotación y el *flip* no se realizan a cada una de las muestras, si no que se realizan de aleatoriamente al 50% de las muestras.

3.2.2.6. Función de pérdida

Al igual que el Data Augmentation, la función de pérdida no se ha modificado en los entrenamientos realizados, pero es importante destacar que ha sido MSE (Error Cuadrático Medio) la función usada. Esta medirá el promedio de los errores al cuadrado siguiendo la siguiente ecuación:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - y'_i)^2$$

Donde y_i es el resultado real esperado y y'_i es la predicción del modelo.

3.3. Métrica de evaluación

Con los modelos entrenados, la siguiente fase del sistema consiste en evaluarlos. La métrica de evaluación que se ha elegido para este sistema ha sido Intersection Over Union [25] (IoU), que nos permite conocer el porcentaje de superposición entre dos regiones. Más detalladamente, en nuestro proyecto nos permitirá medir que porcentaje de la segmentación o las segmentaciones predichas por el modelo de machine learning se corresponden con la segmentación o las segmentaciones etiquetadas reales del conjunto de muestras. De esta forma, cuanto mayor sea el número de pixeles en los que coinciden, mayor será el IoU y, por lo tanto, mejor será el modelo evaluado.

La fórmula que sigue esta métrica de evaluación es la siguiente:

$$IoU = \frac{Área\ de\ Intersección}{Área\ de\ Unión}$$

En la Figura 26 se puede observar cómo sería el caso de la superposición entre la etiqueta de una de las muestras del *dataset* y una predicción.

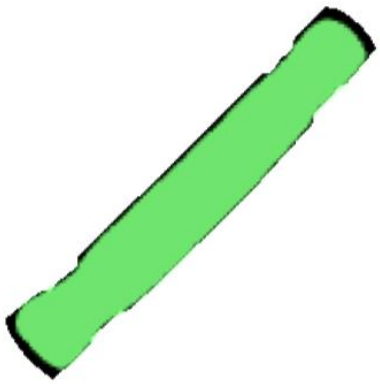


Figura 26. Comparación de áreas para IoU.

[25] https://learnopencv.com/intersection-over-union-iou-in-object-detection-and-segmentation/

3.4. Experimentos y conclusiones

Una vez explicadas las herramientas y el funcionamiento del sistema de detección del objeto más accesible, vamos a ver los resultados de todas las pruebas y entrenamientos realizados en la siguiente Tabla.

Hay que destacar antes de pasar con ellos que, en todos los entrenamientos realizados en el proyecto, la distribución de muestras de entrenamiento y de validación ha sido de un 80% y 20% respectivamente. Los demás parámetros usados han sido explicados en el apartado 3.2.2.

Tabla 1. Resultados de los modelos de detección de objetos más accesibles.

Arquitectura	N.º muestras de entrenamiento	Entrada	Batch Size	Optimizador	Capas de Encoder/ Decoder	IOU (Val)
U-net	250	RGBD	32	Adam	5	**0.932**
U-net	200	RGBD	32	Adam	5	0.875
U-net	100	RGBD	32	Adam	5	0.793
U-net	250	RGBD	32	SGD	5	0.835
U-net	250	RGBD	16	Adam	5	0.911
U-net	250	RGBD	32	Adam	4	0.769
U-net	250	RGB	32	Adam	5	0.920
U-net	250	RGB	16	Adam	5	0.914
Seg-net	250	RGBD	32	Adam	5	0.912
Seg-net	200	RGBD	32	Adam	5	0.886
Seg-net	250	RGBD	16	Adam	5	0.896
Seg-net	250	RGBD	32	SGD	5	0.917
Seg-net	250	RGB	32	Adam	5	0.902
Seg-net	250	RGB	32	SGD	5	0.911
Res-Unet	250	RGBD	32	Adam	5	0.915
Res-Unet	250	RGBD	16	Adam	5	0.910
Res-Unet	250	RGB	32	Adam	5	0.919
Res-Unet	250	RGB	32	SGD	5	0.823
Res-Unet	250	RGB	16	Adam	5	0.917

Viendo la Tabla 1 de resultados, podemos obtener las siguientes conclusiones (con el objetivo de aclarar y complementar ciertas explicaciones, en algunas de ellas se aporta una tabla con los resultados de las combinaciones más representativas para las comparaciones, resultados que se extraen de la Tabla 1):

- La U-net ha sido la arquitectura que mejor resultados ha conseguido en general respecto a las demás, para la mayoría de las combinaciones de parámetros utilizadas. La Tabla 2 contiene comparaciones de pruebas en las que se puede apreciar como Seg-Net y Res-Unet obtienen peor resultado.

Tabla 2. Resultados para comparación de arquitecturas.

Arquitectura	N.º muestras de entrenamiento	Entrada	Batch Size	Optimizador	Capas de Encoder/ Decoder	IOU (Val)
Seg-net	250	RGBD	32	Adam	5	0.912
U-net	250	RGBD	32	Adam	5	0.932
Res-Unet	250	RGBD	32	Adam	5	0.915
Seg-net	250	RGB	32	Adam	5	0.902
U-net	250	RGB	32	Adam	5	0.920
Res-net	250	RGB	32	Adam	5	0.919

- Mirando ahora los mejores modelos de cada arquitectura, las mejores Res-Unet y Seg-net consiguen un resultado muy parecido (aunque menor) al de la mejor U-net, pero es importante destacar y comparar el tiempo de entrenamiento en el que lo consiguen. U-net consigue el mejor IoU en 8668.19 segundos, pero, sin embargo, la Seg-net consigue un resultado parecido en tan solo 4132.9 segundos. Esto es un factor importante a tener en cuenta a la hora de elegir si es preferible un resultado un poco más bajo, pero en la mitad de tiempo, o un resultado levemente superior en un tiempo mucho mayor. En cuanto a Res-Unet, el tiempo de entrenamiento aumenta considerablemente debido a su arquitectura, siendo este de 13996.28 segundos.

Tabla 3. Resultados para comparación de mejores modelos por arquitectura.

Arquitectura	N.º muestras	Entrada	Batch Size	Optimizador	Capas de Encoder/ Decoder	IOU (Val)	Tiempo entren. (seg)
Seg-net	250	RGBD	32	Adam	5	0.912	4132.9
U-net	250	RGBD	32	Adam	5	0.932	8668.19
Res-Unet	250	RGB	32	Adam	5	0.919	13996.28

- Con distintos números de muestras de entrenamiento (teniendo en cuenta que la proporción 80% y 20% para entrenamiento y validación respectivamente), se han realizado varias pruebas para U-net y Seg-net. Estas permiten observar cómo en todos los casos los resultados de IoU obtenidos aumentan cuando incrementamos el número de muestras de entrenamiento.

Tabla 4. Resultados con distintos números de muestras de entrenamiento.

Arquitectura	N.º muestras de entrenamiento	Entrada	Batch Size	Optimizador	Capas de Encoder/ Decoder	IOU (Val)
U-net	250	RGBD	32	Adam	5	0.932
U-net	200	RGBD	32	Adam	5	0.875
U-net	100	RGBD	32	Adam	5	0.793
Seg-net	250	RGBD	32	Adam	5	0.912
Seg-net	200	RGBD	32	Adam	5	0.886

- Un *batch size* de 32 ha sido el usado en la mayoría de los experimentos. Tras el uso de un *batch size* de 16 con varias combinaciones de parámetros se puede ver que, para todas ellas se ha extraído un resultado peor que con el tamaño de *batch* de 32. Por ejemplo, para la mejor combinación de parámetros con U-net, cambiando el *batch size* a 16 el IoU ha ido de 0.911, que supone una leve bajada frente al 0.932 con Batch Size 32.

Tabla 5. Resultados para comparación de Batch Size.

Arquitectura	N.º muestras de entrenamiento	Entrada	Batch Size	Optimizador	Capas de Encoder/ Decoder	IOU (Val)
U-net	250	RGBD	32	Adam	5	0.932
U-net	250	RGBD	16	Adam	5	0.911
U-net	250	RGB	32	Adam	5	0.920
U-net	250	RGB	16	Adam	5	0.914
Seg-net	250	RGBD	32	Adam	5	0.912
Seg-net	250	RGBD	16	Adam	5	0.896
Res-Unet	250	RGB	32	Adam	5	0.919
Res-Unet	250	RGB	16	Adam	5	0.917

- Respecto a los optimizadores, se puede observar que en U-net y Res-Unet el uso de Adam ha resultado mejor que el uso de SGD notablemente (0.932 y 0.919 frente a 0.835 y 0.823). En Seg-net en cambio se ha obtenido un resultado, aunque muy parecido, levemente superior para el optimizador SGD (0.911 y 0.917, frente a 0.902 y 0.912 con Adam).

Tabla 6. Resultados para comparación de optimizadores.

Arquitectura	N.º muestras de entrenamiento	Entrada	Batch Size	Optimizador	Capas de Encoder/ Decoder	IOU (Val)
U-net	250	RGBD	32	Adam	5	0.932
U-net	250	RGBD	32	SGD	5	0.835
Seg-net	250	RGB	32	Adam	5	0.902
Seg-net	250	RGB	32	SGD	5	0.911
Seg-net	250	RGBD	32	Adam	5	0.912
Seg-net	250	RGBD	32	SGD	5	0.917
Res-Unet	250	RGB	32	Adam	5	0.919
Res-Unet	250	RGB	32	SGD	5	0.823

- También se puede concluir que la idea de introducir un cuarto canal de entrada con la codificación de la profundidad junto a la imagen RGB ha sido una buena aportación, ya que el uso de este tipo de entrada ha incrementado levemente el mejor IoU conseguido en cada caso, excepto en el de la Res-Unet, donde el resultado ha sido muy parecido, aunque levemente superior con la entrada de 3 canales. Es importante destacar aquí que el tiempo de entrenamiento para una red con entrada de 4 canales será mayor que el tiempo de la misma red (con los mismos parámetros) con entrada de 3 canales. Esto se debe a que la entrada a procesar es mayor, y por lo tanto requiere un mayor tiempo de cómputo. Por ejemplo, para la mejor U-net con entrada RGBD el tiempo de entrenamiento es de 8668.19 segundos, mientras que con entrada RGB es de 7671.92 segundos.

Tabla 7. Resultados para comparación de entradas.

Arquitectura	N.º muestras de entrenamiento	Entrada	Batch Size	Optimizador	Capas de Encoder/ Decoder	IOU (Val)
U-net	250	RGBD	32	Adam	5	0.932
U-net	250	RGB	32	Adam	5	0.920
Seg-net	250	RGBD	32	Adam	5	0.912
Seg-net	250	RGB	32	Adam	5	0.902
Res-Unet	250	RGBD	32	Adam	5	0.915
Res-Unet	250	RGB	32	Adam	5	0.919

- Y, para terminar, el número de capas de codificador/decodificador en las arquitecturas con mejor resultado ha sido el común de las mismas (que podemos encontrar en sus documentaciones), 5 capas como podemos ver en las Figuras 5, 6 y 25. La prueba con una capa menos en U-net ha resultado en un IoU mucho menor, del 0.769.

Tabla 8. Resultados para comparación de número de capas de codificador/decodificador.

Arquitectura	N.º muestras de entrenamiento	Entrada	Batch Size	Optimizador	Capas de Encoder/ Decoder	IOU (Val)
U-net	250	RGBD	32	Adam	5	0.927
U-net	250	RGBD	32	Adam	4	0.769

Para los modelos con los 2 mejores resultados de la tabla, se ha decido realizar una validación cruzada con el fin de obtener un resultado final más detallado de los mismos, y así poder llevar a cabo una mejor comparación.

La validación cruzada[26] es un método de evaluación de los resultados de un análisis estadístico, muy utilizada en proyectos de inteligencia artificial para validar modelos generados, que permite garantizar la independencia entre el subconjunto de datos de entrenamiento y el de prueba. Consiste en iterar y calcular la media aritmética de las precisiones de predicción de las diferentes particiones. Esta técnica se usará en casos donde el principal fin es la predicción y se desea estimar la precisión de un modelo que se llevará a cabo a la práctica, como es el caso de este trabajo. El tipo de validación cruzada que se va a llevar a cabo en esta memoria es la de K iteraciones, siendo K igual a 10 para nuestro conjunto de datos.

La validación cruzada k-fold divide el conjunto de datos en K subconjuntos de igual tamaño de forma aleatoria. De esta forma, el modelo utiliza el primer subconjunto en la primera iteración para probar el modelo, usando los conjuntos de datos restantes para el entrenamiento. El mismo proceso se repite hasta que se utilizan todos los subconjuntos como conjunto de validación. Finalmente, para obtener el resultado definitivo se calculará la media aritmética de los resultados de cada iteración. Este método es muy preciso, ya que la evaluación se realiza a partir de K combinaciones de datos de entrenamiento y de prueba, pero a pesar de esto tiene la desventaja de que puede resultar lento desde el punto de vista computacional. Es por esto último por lo que únicamente se ha realizado esta prueba para los mejores modelos obtenidos en las pruebas iniciales, ya que esta validación puede durar entre 10 y 15 horas dependiendo de los parámetros establecidos.

Los resultados de esta validación cruzada se pueden ver en la siguiente Tabla (9).

[26] https://es.wikipedia.org/wiki/Validación_cruzada

Tabla 9. Resultados de validación cruzada para los mejores modelos.

Modelo	IoU (Validación cruzada)
U-Net/250/RGBD/32/Adam/5	(0.918 + 0.932 + 0.917 + 0.916 + 0.930 + 0.938 + 0.932 + 0.941 + 0.912 + 0.929) / 10 = **0.927**
U-Net/250/RGB/32/Adam/5	(0.919 + 0.928 + 0.901 + 0.924 + 0.911 + 0.931 + 0.918 + 0.929 + 0.931 + 0.915) / 10 = **0.9207**

Se puede concluir viendo estos dos resultados que, realizando la validación cruzada, el modelo que mejor precisión obtiene es el mismo que en las anteriores pruebas, obteniendo esta vez una media de 0.927 de IoU de las 10 iteraciones realizadas. Como se puede observar en la tabla, el máximo IoU obtenido por este modelo en una iteración para las muestras de validación ha sido de un **0.941**, lo cual es un resultado muy favorable.

3.5. Análisis del mejor modelo

Como se puede observar en la tabla de resultados del apartado anterior, el mejor modelo consta de una red neuronal convolucional con arquitectura U-net, entrenada con 250 muestras, de las cuales el 80% han sido para entrenamiento y el 20% para validación, cuya entrada cuenta con 4 canales (de los cuales 3 se corresponden con los de la imagen RGB y 1 con la profundidad), con un Batch Size de 32 muestras, optimizador Adam y 5 capas de codificador/decodificador (4 skip-connections). A continuación, vamos a ver los resultados obtenidos por este modelo de forma más detallada, y se comentarán algunos aspectos a destacar sobre ellos.

En la Figura 27 se pueden observar las gráficas de *loss* e IoU obtenidas por el modelo, que mostrarán como avanzan ambos resultados a lo largo de las 10 *stages* del entrenamiento (es decir, que resultados se obtienen cada 100 *epochs*).

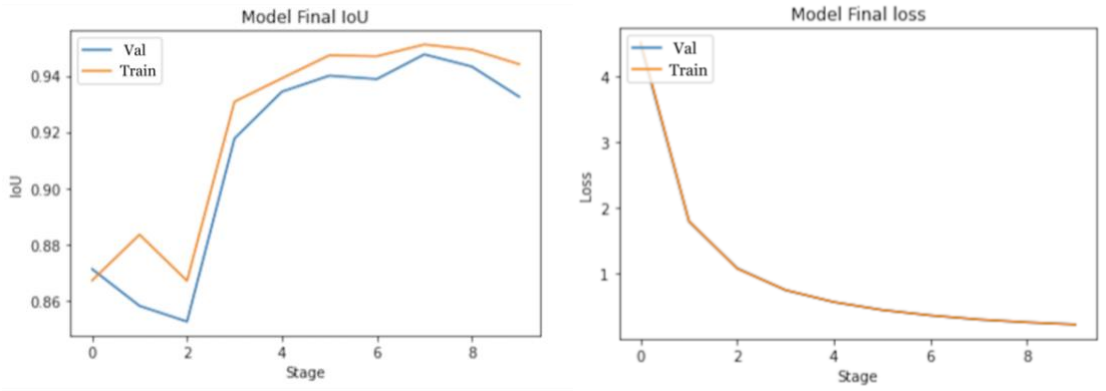


Figura 27. Gráficas de resultados del mejor modelo.

Respecto al IoU, se puede ver que el mejor resultado en validación se ha obtenido tras el entrenamiento de la *stage* 7, es decir, tras 700 *epochs*, que incluso llega a superar 0.940 de IoU. Sin embargo, en la gráfica del *loss* podemos ver como este sí que sigue bajando a lo largo de las *stages*. El mínimo error se ha obtenido tras el entrenamiento de la última *stage*.

Además de las gráficas, y para obtener una idea más visual de cómo evoluciona este mejor modelo, en la siguiente Figura (28) se va a observar cómo las predicciones de segmentación van mejorando a lo largo de las *stages* del entrenamiento, obteniendo cada vez una estructura del objeto más parecida a la real.

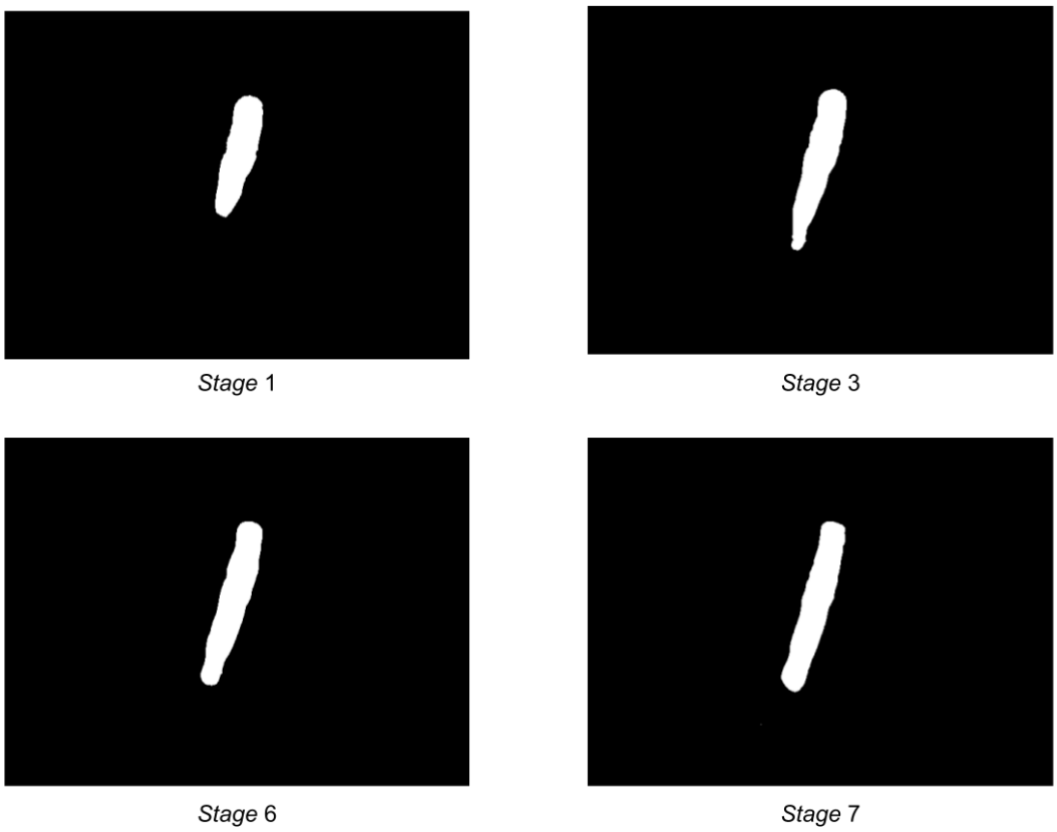


Figura 28. Evolución de las predicciones para el mejor modelo.

Y, para finalizar, es importante comentar que el tiempo de entrenamiento de este modelo ha sido 8668.19 segundos, lo que equivale a 2 horas y 24 minutos.

4. Extracción del mejor agarre

Con el objeto más accesible detectado entre todos los captados por la cámara, el sistema debe ahora extraer su agarre. El agarre cuenta con 5 características que lo diferencian: el centro (coordenada "x", coordenada "y"), la anchura, el ángulo de giro y la orientación del ángulo de giro. Esta sería la información necesaria y suficiente que se debería pasar al robot para que este pudiese realizar la acción de agarrar el objeto.

El sistema que se encarga de extraer estos datos se va a explicar paso a paso en los diferentes apartados de este capítulo, comentando además la metodología usada para cada una de las fases. Tras esto, en el último apartado se realizará una evaluación del sistema completo de extracción del mejor agarre, con el objetivo de conocer con cuál de los modelos entrenados en el capítulo anterior, se obtienen unas características de agarre más precisas o parecidas a las reales.

4.1. Contornos

Para poder calcular cada una de las características del agarre del objeto más accesible, lo primero que se va a necesitar es extraer el contorno del mismo.

Pero, antes de ir con la detección de contornos, es necesario establecer el umbral de la imagen segmentada, lo que podemos hacer usando las funciones de la librería OpenCV de Python `cvtColor()` para convertir la imagen a grises, `GaussianBlur()` para difuminar y `threshold()` para aplicar la umbralización (si el valor del píxel es menor que el umbral, se establece en 0; de lo contrario, se establece en un valor máximo).

Una vez aplicadas estas tres funciones, mediante la segmentación umbralizada y la función `findContours()` de la librería OpenCV de Python se podrá extraer el contorno exacto del objeto segmentado más accesible (si en una imagen hubiese más de un objeto accesible, se elegiría el contorno de aquel cuya posición sea más alta, es decir, los píxeles en la imagen de profundidad tienen menor valor). Será sobre este contorno (representado en Python como una lista de puntos) sobre el que se trabajará para calcular las características del agarre. En la Figura 29 podemos ver un ejemplo para la predicción de la muestra 108 del *dataset* de uno de nuestros modelos de extracción del objeto más accesible.

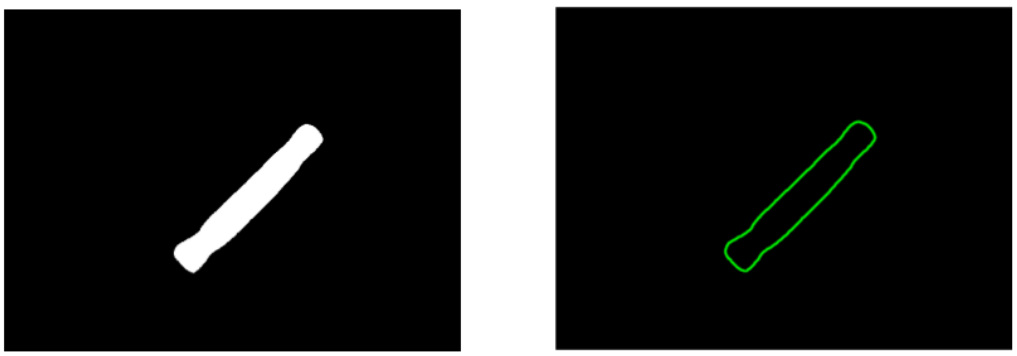


Figura 29. Contorno del objeto.

4.2. Punto central

La primera de las características que se va a extraer del agarre es su punto central, que coincidirá con el punto central del objeto. Se va a conseguir con la función `moments()` de OpenCV, con la que se extraerá el momento de la imagen para el contorno. Un momento de imagen es un promedio ponderado particular de las intensidades de píxeles de la imagen, con la ayuda del cual podemos encontrar algunas propiedades específicas de una imagen como, por ejemplo, radio, área, centroide, etc.

En particular, el centro del contorno se puede extraer siguiendo las siguientes fórmulas:

$$c_x = \frac{M10}{M00}$$

$$c_y = \frac{M01}{M00}$$

donde c_x y c_y son, respectivamente, las coordenadas "x" e "y" del punto central y M es el momento.

En la Figura 30, podemos ver el centro dibujado con un punto blanco dentro del contorno de la segmentación.

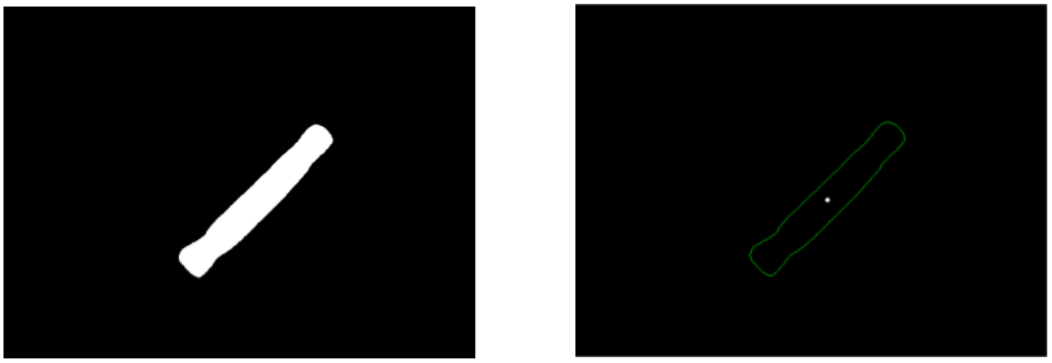


Figura 30. Punto central del agarre.

4.3. Anchura del agarre

Tras el centro, la anchura del agarre se va a extraer mediante el corte entre la línea del contorno y la recta del eje del objeto correspondiente a su lado más corto que pasa por su centro. La distancia entre estos dos puntos de corte será la anchura del agarre.

El primer paso en esta fase será extraer el eje del lado del objeto con mayor longitud, lo cual se puede realizar mediante la función `fitLine()`, que ajusta todo el conjunto de puntos del contorno a esta recta. Con ella y trabajando con las ecuaciones de las rectas (y con los valores de los vectores), se podrá conocer su perpendicular, que coincidirá con el eje correspondiente al lado corto del objeto.

Teniendo esto, la función `where()` de Numpy será la que permita extraer los puntos de corte, que son aquellos píxeles donde coinciden los puntos de la recta y el contorno.

Y, finalmente, la función `distance.euclidean()` de la librería `Spicy` nos devolverá la distancia euclídea en píxeles entre ambos puntos de corte.

En la Figura 31 se puede ver el eje correspondiente al lado de menor longitud del objeto en color azul claro y la anchura del agarre, que se corresponde con la distancia entre los puntos de corte de este con el contorno.

Figura 31. Anchura del agarre.

4.4. Ángulo y orientación

Para estas dos características, partimos del eje correspondiente al lado corto del objeto obtenido en el apartado anterior. Será su ángulo respecto a los ejes de la imagen el que nos permita conocer el giro que el robot debe realizar para el correcto agarre de la pieza.

El primer paso será extraer los vectores unitarios tanto del eje de la pieza como del eje de la imagen sobre el que lo vamos a comparar para extraer el ángulo, que en este caso será el eje Y (vertical). Ambos vectores unitarios se conseguirán mediante la división del vector por su norma (que extraemos mediante la función `linealg.norm()` de Numpy). Ahora, el arcocoseno (`np.arccos()`) del producto escalar (`np.dot()`) de ambos vectores unitarios va a ser el ángulo en radianes del agarre. El signo de este ángulo va a ser el que nos defina la orientación del agarre (signo negativo significará que el giro es hacia la derecha y signo positivo hacia la izquierda).

En la Figura 32 se puede ver el ángulo que se crea entre el eje vertical de la imagen y el eje correspondiente al lado de menor longitud del objeto. La orientación para este ejemplo sería hacia la izquierda.

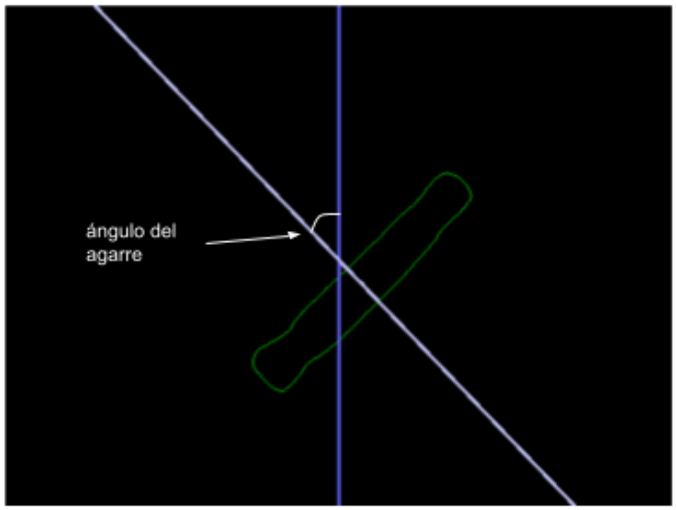


Figura 32. Ángulo y Orientación del agarre.

4.5. Evaluación con varianza

Para evaluar la precisión del agarre de nuestro sistema completo (sistema de detección del objeto más accesible junto al sistema de detección del mejor agarre) se ha propuesto la siguiente solución. Esta consistirá en una varianza entre las características del agarre obtenidas para la segmentación real (etiqueta del conjunto de muestras) y las obtenidas para la segmentación predicha del objeto elegido como más accesible (como se ha comentado, el más alto de entre los detectados como accesibles) por los modelos del capítulo anterior. Esta varianza se realizará para cada una de las muestras y, finalmente, se hará una media del total para cada modelo. Cuanto menor sea esta varianza media, mejor será el sistema.

De esta forma, se podrá evaluar el sistema total no solo mediante el IoU de los modelos de detección del objeto más accesible (es decir, por el porcentaje de coincidencia de pixeles de la etiqueta con la predicción), sino que también se evaluará la correcta estructura formada por estos píxeles, buscando que esta sea lo más parecida posible a la estructura del objeto real. Cuanto más parecidas (con menos varianza) sean las características de agarre de la segmentación predicha respecto a las de la segmentación real, más parecida deberían ser las estructuras.

En la Tabla 10 podremos ver esta evaluación para las predicciones de 5 de los mejores modelos de detección de objetos accesibles obtenidos en el capítulo anterior.

Tabla 10. Resultados de las varianzas en características de agarre.

Modelo	Varianza Centro (x,y) (píxeles)	Varianza Anchura (píxeles)	Varianza (Ángulo) (radianes)
U-Net/250/RGBD/32/Adam/5 (0.932 IoU)	(1.99, 2.39)	2.52	0.03 rad= 1.71º
U-Net/250/RGB/32/Adam/5 (0.920 IoU)	(1.79, 2.13)	2.47	0.023 rad = 1.32 º
Res-Unet/250/RGB/32/Adam/5 (0.919 IoU)	(1.13, 1.19)	1.76	0.022 rad= 1.26 º
Res-Unet/250/RGBD/32/Adam/5 (0.915 IoU)	(0.99, 1.22)	1.46	0.021 rad = 1.2 º
Seg-Net/250/RGBD/32/Adam/5 (0.912 IoU)	(1.65, 1.70)	3.39	0.026 rad = 1.5 º

Viendo la Tabla 10, es importante comentar algunas conclusiones. A pesar de los resultados de IoU extraídos en el capítulo anterior, a la hora de extraer el mejor agarre y que este coincida con el real, los modelos que resultaban ser más precisos pueden resultar no serlo en este caso. Esto es lo que ocurre con los dos modelos entrenados con arquitectura de U-net que, como podemos ver en la Tabla 1, tienen los valores de IoU más altos, pero a pesar de ello, a la hora de evaluar las características de agarre obtienen las varianzas más altas de los 5 modelos comparados.

El modelo con Seg-net obtiene un resultado parecido o un poco mejor en algunas características respecto a U-net, pero los más precisos en este caso son los modelos con arquitectura Res-Unet, en concreto aquel entrenado con entrada de 4 canales y 250

muestras de entrenamiento. Como se puede ver en la tabla de resultados, la varianza media respecto al punto central es de 0.99 pixeles en la coordenada “x” y 1.22 en la “y”, la anchura solamente tiene una varianza de 1.46 y el ángulo de 1.2 grados. Por lo tanto, buscando la mejor precisión del sistema completo, nos quedaríamos con este modelo con Res-Unet.

5. Conclusiones

Una vez finalizado el desarrollo del software y la fase de pruebas del sistema, así como su explicación en la memoria, se pueden extraer varias conclusiones.

En el apartado 1.2 se proponen unos objetivos claros para este proyecto: el desarrollo de un sistema de reconocimiento del objeto más accesible para un brazo robótico y el desarrollo de un sistema que detecte el mejor agarre para este objeto detectado. A pesar de que la calidad de los resultados no es perfecta, el balance general del proyecto ha sido positivo, ya que estos objetivos se han alcanzado con éxito.

Para esto, en primer lugar, se ha construido un *dataset* de 250 muestras desde cero, enfocado a la segmentación de aquellos objetos de una imagen que sean más accesibles por un brazo robótico. A pesar de no ser un conjunto de datos de gran tamaño y muy variado, ya que únicamente contiene muestras para la segmentación de un tipo de objeto (tuberías de pequeño tamaño), se puede considerar un conjunto de datos importante y completo para el objetivo de este proyecto. Por otro lado, gracias a este *dataset* se ha podido desarrollar el sistema de redes neuronales convolucionales, el cual permite posteriormente para una muestra, conocer que objeto de esta podría ser agarrado por el brazo robótico. Como se ha podido ver en la sección 3.5, la precisión del mejor modelo no es del 100%, pero sí es mayor al 90%, lo cual ha sido suficiente para que un buen tratamiento de los resultados permita el correcto funcionamiento del sistema de detección del mejor agarre desarrollado posteriormente.

Con este último, se han logrado obtener características de agarre muy parecidas a las reales (con variaciones mínimas en los ángulos y distancias calculadas), y esto es debido en su totalidad a los buenos resultados que los modelos anteriores han obtenido. En el apartado 4.5 se puede ver cómo no siempre obtener el mejor IoU significa obtener el agarre más preciso (aunque sí que está muy relacionado), y es por esto por lo que se ha realizado una evaluación del sistema completo mediante varianzas en las características del agarre, con el objetivo de ver cuál es realmente aquel modelo que extrae los agarres más precisos.

La precisión obtenida podría incrementar aumentando el *dataset* actual con muestras etiquetadas correctamente o incluso con entrenamientos con más epochs para los modelos de extracción del objeto más accesible que quizás puedan alcanzar un IoU mayor. También sería favorable probar nuevas arquitecturas de modelos de redes neuronales convolucionales que a día de hoy se van desarrollando y que posiblemente respondan mejor que las implementadas en este proyecto.

Respecto al entrenamiento de este primer sistema de reconocimiento del objeto más accesible, es importante concluir que los entrenamientos de este tipo de redes neuronales convolucionales requieren de una gran capacidad de cómputo en las máquinas usadas. En este proyecto, se ha trabajado con 2 tarjetas gráficas NVIDIA RTX 3090 24GB DDR4, que son actualmente muy potentes y gracias a las cuales el tiempo de entrenamiento de las pruebas para el proyecto se ha reducido considerablemente. Sin embargo, a pesar de esto, los entrenamientos más cortos (con menos datos) se han alargado hasta las 3 horas. Es por ello que para poder realizar un proyecto de estas características y con una cantidad de pruebas considerable, va a ser casi necesario contar con buenas infraestructuras de cómputo y elegir eficientemente las pruebas y entrenamientos que se quieren realizar, para evitar perder tiempo y recursos en entrenamientos que no aportarán nada al proyecto.

Y, para finalizar con las conclusiones y con el objetivo de contextualizar las restricciones temporales que tendrían las aplicaciones en las que use el sistema, se va a comentar el tiempo de cómputo que implica la detección del agarre en tiempo real. Este tiempo se ha obtenido sumando el tiempo de generación de la máscara del objeto más accesible con el mejor modelo entrenado de todo el proyecto y el tiempo de cálculo del agarre. Tras 50 pruebas, se ha comprobado que el tiempo medio que el sistema tarda en predecir el objeto más accesible de una muestra es de 0.18 segundos, mientras que el tiempo medio de extracción de las características de agarre es de 0.007 segundos. La suma de ambas medias resulta de 0.187 segundos, que es lo suficientemente bajo para confirmar que el sistema se podría aplicar en diversas situaciones a tiempo real.

5.1. Relación del proyecto con los estudios cursados

A lo largo del máster se han estudiado materias que han permitido y ayudado al desarrollo de este trabajo. La mayoría de ellas han aportado al proyecto de alguna forma, desde asentando las bases de la inteligencia artificial o *machine learning*, hasta concediendo conocimientos más especializados en redes neuronales convolucionales para segmentación semántica. A continuación, se van a mencionar algunas de ellas por su gran aportación.

En primer lugar, se destacan asignaturas del primer módulo como RFA (Reconocimiento de Formas y Aprendizaje Computacional) o TIA (Técnicas de Inteligencia Artificial), las cuales han permitido establecer una gran base en el aprendizaje del alumno en la inteligencia artificial y en su posible uso en herramientas y aplicaciones de la vida real.

En el segundo módulo del máster, HAIA (Herramientas y Aplicaciones de la Inteligencia Artificial), RNA (Redes Neuronales Artificiales) y PEE (Predicción Estructurada Estadística) han establecido las bases del funcionamiento de las redes neuronales y el *deep learning*, permitiendo al alumno familiarizarse con su integración en diversos proyectos. No obstante, se destaca RNA debido a que ha sido la fuente principal para la consolidación de los principios del tratamiento de las redes neuronales en cada una de sus posibles aplicaciones.

Y, por último, del tercer módulo cabe destacar VPC (Visión por Computador), que ha ido más allá de las bases en el uso del *deep learning* para problemas de visión artificial. Esta asignatura ha permitido al alumno hacer uso del DL en tareas como el reconocimiento de objetos, cambio de estilos de imágenes, completar imágenes o, como es el caso de este proyecto, segmentar imágenes. Por lo tanto, ha sido esta asignatura la que ha permitido conocer el funcionamiento de la arquitectura de redes neuronales en forma de “u”, con codificador y decodificador.

6. Trabajos futuros

Una vez visto todo el desarrollo realizado y las conclusiones extraídas del mismo, se comentan posibles trabajos y ampliaciones futuras que se podrían aplicar a este proyecto.

La primera de ellas podría tratarse, como se ha comentado en las conclusiones, de un aumento del conjunto de datos. Un aumento no solo del número de muestras con diferente disposición del tipo de objeto con el que contamos actualmente, si no con nuevos tipos de objetos que puedan ser agarrados por el brazo robótico. De esta forma, el sistema podría detectar formas diferentes y segmentarlas para posteriormente obtener su agarre. Para esta ampliación, sería interesante probar a introducir un modelo previo al de segmentado, que ayudase a reconocer que tipo de objeto de entre los posibles es el que se está viendo por cámara, para que así luego el modelo de segmentado tenga menos dificultad a la hora de extraer la estructura segmentada del mismo. Con esto, el entrenamiento sería mucho más costoso debido al mayor número de muestras y el nuevo modelo de detección de objetos, pero se trataría de un sistema mucho más robusto y polivalente.

En cuanto al sistema de detección del mejor agarre, sería interesante hacer uso del machine learning para crear un modelo que detecte los agarres automáticamente. El sistema de este proyecto funciona correctamente trabajando con la estructura del objeto, pero sería conveniente para posibles nuevos objetos entrenar modelos que detecten la mejor forma de agarrar cada uno de ellos. Existen objetos que posiblemente no puedan ser agarrados en su totalidad debido a su tamaño, pero sin embargo con un modelo correctamente entrenado, el robot podría agarrarlo de un punto determinado y moverlo correctamente. Un ejemplo sería una taza, la cual quizás el brazo robótico no podría agarrar completamente, pero sí podría hacerlo de su asa. Para esta ampliación, se requeriría de un nuevo *dataset* con imágenes de los distintos objetos (como *Cornell Grasp Dataset* en el apartado 2.4), con posibles características de agarre como las que extraemos con el sistema de detección de agarre de este proyecto (centro, anchura, ángulo y orientación del agarre) como etiquetas. Con este conjunto de datos y redes neuronales convolucionales bien entrenadas podríamos tener un sistema que, junto al anterior de segmentado, funcione de forma muy efectiva.

Y, para terminar con las posibles ampliaciones, una de las más importantes sería probar el sistema al completo en escenarios industriales reales. Mediante la comunicación con el brazo robótico, se probaría que los ángulos y distancias que se detectan con los modelos entrenados funcionan correctamente y se comprobaría que este sistema se puede poner en marcha en entornos industriales.

Bibliografía

[1] La Serna Palomino, N., & Román Concha, U. N. (2009). Técnicas de Segmentación en Procesamiento Digital de Imágenes. Revista De investigación De Sistemas E Informática, 6(2), 9–16.

[2] La Serna Palomino, N., & Luxmila Pró Concepción (2010). Waterseh: un algoritmo eficiente y flexible para segmentación de imágenes de geles 2-DE. Revista De investigación De Sistemas E Informática, 7(2), 35–41.

[3] Ramirez, E., Martínez, D., & Carmona, R. (2012). Segmentación de imágenes a color basada en el algoritmo de GrabCut. Tekhné, (15).

[4] Chen, P. C., & Pavlidis, T. (1979). Segmentation by texture using a co-occurrence matrix and a split-and-merge algorithm. Computer graphics and image processing, 10(2), 172-182.

[5] Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 3431-3440).

[6] Badrinarayanan, V., Kendall, A., & Cipolla, R. (2017). Segnet: A deep convolutional encoder-decoder architecture for image segmentation. IEEE transactions on pattern analysis and machine intelligence, 39(12), 2481-2495.

[7] Ronneberger, O., Fischer, P., & Brox, T. (2015, October). U-net: Convolutional networks for biomedical image segmentation. In International Conference on Medical image computing and computer-assisted intervention (pp. 234-241). Springer, Cham.

[8] Chen, L. C., Papandreou, G., Kokkinos, I., Murphy, K., & Yuille, A. L. (2017). Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. IEEE transactions on pattern analysis and machine intelligence, 40(4), 834-848.

[9] Chen, L. C., Zhu, Y., Papandreou, G., Schroff, F., & Adam, H. (2018). Encoder-decoder with atrous separable convolution for semantic image segmentation. In Proceedings of the European conference on computer vision (ECCV) (pp. 801-818).

[10] Redmon, J., & Angelova, A. (2015, May). Real-time grasp detection using convolutional neural networks. In 2015 IEEE international conference on robotics and automation (ICRA) (pp. 1316-1322). IEEE.

[11] Kumra, S., & Kanan, C. (2017, September). Robotic grasp detection using deep convolutional neural networks. In 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 769-776). IEEE.

[12] Alom, M. Z., Taha, T. M., Yakopcic, C., Westberg, S., Sidike, P., Nasrin, M. S., ... & Asari, V. K. (2018). The history began from alexnet: A comprehensive survey on deep learning approaches. arXiv preprint arXiv:1803.01164.

[13] Zagoruyko, S., & Komodakis, N. (2016). Wide residual networks. arXiv preprint arXiv:1605.07146.

[14] Zhang, Z., Liu, Q., & Wang, Y. (2018). Road extraction by deep residual u-net. IEEE Geoscience and Remote Sensing Letters, 15(5), 749-753.

[15] BROWNLEE, J. (2018). WHAT IS THE DIFFERENCE BETWEEN A BATCH AND AN EPOCH IN A NEURAL NETWORK. MACHINE LEARNING MASTERY, 20.

[16] RUDER, S. (2016). AN OVERVIEW OF GRADIENT DESCENT OPTIMIZATION ALGORITHMS. ARXIV PREPRINT ARXIV:1609.04747.

[17] ZHANG, Z. (2018, JUNE). IMPROVED ADAM OPTIMIZER FOR DEEP NEURAL NETWORKS. IN 2018 IEEE/ACM 26TH INTERNATIONAL SYMPOSIUM ON QUALITY OF SERVICE (IWQOS) (PP. 1-2). IEEE.