Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Máster Universitario en Ingeniería Informática

Trabajo Fin de Máster

# REDES NEURONALES APLICADAS A LA SEGMENTACIÓN SEMÁNTICA

Autor: Lucas Martín Gil-Delgado
Tutor: Javier de Lope Asiaín

Madrid, Junio 2025

# Resumen

Dentro del ámbito de la visión por ordenador, la segmentación de imágenes juega un papel esencial al permitir descomponer cada escena en piezas significativas: desde carreteras y edificios hasta peatones y vehículos. Estos métodos permiten, por ejemplo, a la agricultura de precisión monitorizar el estado de los cultivos y anticipar posibles incidencias, así como a los sistemas de conducción autónoma identificar en tiempo real elementos críticos de la vía.

En este trabajo, se estudian tres arquitecturas de referencia: U-Net y DeepLabV3+ para segmentación semántica, y Mask R-CNN para segmentación por instancia, para contrastar con la semántica. En primer lugar, se realiza una revisión exhaustiva del estado del arte, identificando los fundamentos teóricos y las bases que han contribuido a la eficacia de dichos modelos. A continuación, se lleva a cabo la selección y el preprocesado de un conjunto de datos público.

La implementación se desarrolla mediante PyTorch y TensorFlow, aprovechando sus funcionalidades para configurar *pipelines* de entrenamiento modulares. Se ajustan hiperparámetros críticos y se incorporan técnicas avanzadas de aumento que mejoran la capacidad de generalización. Para una evaluación objetiva, se emplean métricas como mIoU (*mean Intersection over Union*) o *validation accuracy*, además de que se hace constar el consumo de recursos computacionales de cada modelo.

Los resultados obtenidos muestran que cada arquitectura presenta un equilibrio distinto entre velocidad y exactitud: U-Net destaca por su rapidez en contextos donde se requiere procesamiento ágil, DeepLabV3+ ofrece mayor precisión en el contorno de objetos complejos, y Mask R-CNN demuestra una efectividad superior en la separación de instancias. Finalmente, se extraen recomendaciones para la selección del modelo más apropiado según criterios de precisión, velocidad y disponibilidad de recursos.

# Abstract

Within the field of computer vision, image segmentation plays an essential role by allowing each scene to be decomposed into meaningful parts: from roads and buildings to pedestrians and vehicles. These methods enable, for example, precision agriculture to monitor crop health and anticipate potential issues, as well as autonomous driving systems to identify critical roadway elements in real time.

In this work, three reference architectures are studied: U-Net and DeepLabV3+ for semantic segmentation, and Mask R-CNN for instance segmentation, in order to contrast it with semantic approaches. First, a comprehensive review of the state of the art is conducted, identifying the theoretical foundations and principles that have contributed to the effectiveness of these models. Next, the selection and preprocessing of a public dataset are carried out.

The implementation is developed using PyTorch and TensorFlow, leveraging their capabilities to configure modular training pipelines. Critical hyperparameters are tuned and advanced augmentation techniques are incorporated to improve generalization capacity. For objective evaluation, metrics such as mIoU (mean Intersection over Union) and validation accuracy are employed, and the computational resource consumption of each model is also reported.

The results obtained show that each architecture presents a different balance between speed and accuracy: U-Net stands out for its speed in contexts requiring agile processing, DeepLabV3+ offers greater precision in outlining complex objects, and Mask R-CNN demonstrates superior effectiveness in separating instances. Finally, recommendations are drawn for selecting the most appropriate model according to criteria of accuracy, speed, and resource availability.

# Tabla de contenidos

**1. Introducción ........................................................................ 1**

**2. Estado del arte ........................................................................ 3**

2.1 Estado del arte en segmentación semántica ........................................ 3

2.1.1 Arquitecturas totalmente convolucionales ........................................ 3

2.1.2 Modelos basados en codificador-decodificador ........................................ 3

2.1.3 Arquitecturas basadas en pirámides ........................................ 4

2.1.4 Modelos basados en atención y *transformers* ........................................ 4

2.1.5 Modelos más recientes ........................................ 6

2.2 Estado del arte en segmentación por instancia ........................................ 6

2.2.1 Enfoques basados en propuestas ........................................ 6

2.2.2 Enfoques libres de propuestas ........................................ 8

2.2.3 Modelos basados en *transformers* ........................................ 9

2.2.4 Modelos más recientes ........................................ 10

**3. Fundamentos teóricos ........................................................................ 12**

3.1 Modelos empleados para segmentación Semántica ........................................ 12

3.1.1 U-Net ........................................ 12

3.1.2 DeepLabV3+ ........................................ 14

3.1.2.1 DeepLab v1 ........................................ 14

3.1.2.2 DeepLab v2 ........................................ 17

3.1.2.3 DeepLab v3 ........................................ 18

3.1.2.4 Explicación de DeepLab v3+ ........................................ 19

3.2 Modelos empleados para segmentación por instancia ........................................ 21

3.2.1 Mask R-CNN ........................................ 21

3.2.1.1 R-CNN ........................................ 21

3.2.1.2 Fast R-CNN ........................................ 29

3.2.1.3 Faster R-CNN ........................................ 33

3.2.1.4 Explicación de Mask R-CNN ........................................ 35

**4. Solución propuesta ........................................................................ 40**

4.1 Arquitectura U-Net ........................................ 41

4.1.1 Preprocesado de datos de U-Net ........................................ 41

4.1.2 Arquitectura e hiperparámetros de la implementación U-Net ........................................ 41

4.1.3 Resultados de U-Net ........................................ 42

4.2 Arquitectura DeepLabv3+ ........................................ 45

4.2.1 Arquitectura DeepLabv3+ del artículo original ........................................ 45

4.2.1.1 Preprocesado de datos de DeepLabv3+ ........................................ 45

4.2.1.2 Arquitectura e hiperparámetros de la implementación DeepLabv3+ ........................................ 45

4.2.1.3 Resultados de DeepLabv3+ ........................................ 47

4.2.2 Arquitectura modificada ........................................................... 49
4.2.2.1 Preprocesado de datos de DeepLabv3+ ................................. 49
4.2.2.2 Arquitectura e hiperparámetros de la implementación DeepLabv3+ ........................................................................... 50
4.2.2.3 Resultados de DeepLabv3+ sin preetrenamiento ................... 50
4.2.2.4 Resultados de DeepLabv3+ con preetrenamiento .................. 53
4.3 Arquitectura Mask R-CNN ............................................................. 55
4.3.1 Preprocesado de datos de Mask R-CNN ...................................... 55
4.3.2 Arquitectura e hiperparámetros de la implementación Mask R-CNN 56
4.3.3 Resultados de Mask R-CNN ....................................................... 57
4.4 Comentario de resultados ............................................................. 60
**5. Conclusiones ................................................................................ 62**
**6. Líneas futuras ............................................................................... 64**
**Bibliografía ....................................................................................... 65**
**Apéndice ........................................................................................... 69**
A.1 ...................................................................................................... 69
A.2 ...................................................................................................... 70
A.3 ...................................................................................................... 72
A.4 ...................................................................................................... 74

# 1. Introducción

La inteligencia artificial ha experimentado un crecimiento exponencial en las últimas décadas, revolucionando campos tan diversos como la medicina o la seguridad [1, 2]. En este contexto, el procesamiento de imágenes mediante técnicas de visión por ordenador se ha consolidado como una herramienta fundamental para abordar problemas complejos. Un área especialmente relevante dentro de esta es la segmentación de imágenes, la cual posibilita clasificar y delimitar objetos dentro de una escena.

La segmentación de imágenes se puede abordar desde dos enfoques principales. Por un lado, la segmentación semántica que se centra en clasificar cada píxel de una imagen según su categoría. Por otro lado, la segmentación por instancia, que va más allá, ya que no solo identifica la categoría de cada píxel, sino que también distingue entre diferentes objetos de la misma clase, lo que resulta crucial para identificar, por ejemplo, cada vehículo individual en una escena de tráfico. Estas técnicas son de gran relevancia en aplicaciones críticas como la agricultura de precisión [3] o la conducción autónoma [4]. Sin embargo, implementar estos métodos de manera eficiente implica superar desafíos relacionados con la preservación de detalles espaciales, la gestión de contextos complejos y la diferenciación entre instancias superpuestas.

Para abordar estos retos, han surgido diversas arquitecturas. En el ámbito de la segmentación semántica, han sentado las bases modelos como U-Net y DeepLabV3+.

En lo que respecta a la segmentación por instancia, Mask R-CNN se ha posicionado como el modelo de referencia. Mask R-CNN no solo detecta y clasifica objetos, sino que también genera máscaras precisas que delimitan cada objeto, permitiendo una segmentación detallada y diferenciada.

Este trabajo de fin de máster se centra en el estudio, la implementación y la comparación de estas arquitecturas en problemas de segmentación, evaluando su rendimiento en distintos contextos y escenarios de aplicación. Los subobjetivos del proyecto se articulan de la siguiente manera:

1) **Revisión del estado del arte:** Se realizará un análisis exhaustivo de las bases teóricas y las arquitecturas subyacentes de U-Net, DeepLabV3+ y Mask R-CNN.
2) **Preparación de datos:** Se seleccionará y preprocesará un conjunto de datos público, que permitan entrenar y evaluar los modelos de manera consistente y comparativa.
3) **Implementación y entrenamiento:** Se desarrollarán *pipelines* de entrenamiento utilizando *frameworks* modernos como PyTorch o TensorFlow, optimizando hiperparámetros y empleando técnicas de aumento de datos para mejorar la robustez del modelo.
4) **Evaluación comparativa:** Se utilizarán métricas de rendimiento para contrastar la eficacia de los modelos en tareas de segmentación semántica frente a segmentación por instancia.
5) **Análisis de aplicabilidad:** Se identificarán los escenarios y condiciones en los que cada enfoque es más adecuado, considerando factores como la precisión, la velocidad de procesamiento y los requisitos computacionales.

La estructura del trabajo se organiza en los siguientes capítulos:

- El capítulo 2 detalla el estado del arte de técnicas de segmentación de imágenes, centrándose en los modelos más relevantes en este ámbito.
- El capítulo 3 indica los fundamentos teóricos que subyacen a los tres modelos seleccionados: U-Net, DeepLabV3+ y Mask R-CNN.
- El capítulo 4 lleva a cabo la solución propuesta, explicando cómo se ha implementado cada modelo neuronal a nivel de arquitectura e hiperparámetros, cómo se ha preprocesado el conjunto de datos y los resultados que se han obtenido.
- El capítulo 5 concluye los resultados obtenidos mediante una justificación analítica indicándose las ventajas de cada técnica frente a las otras.
- El capítulo 6 plantea futuros trabajos que se pueden desarrollar a partir de este.

# 2. Estado del arte

El estado del arte se ha desarrollado en dos líneas principales de investigación: la segmentación semántica y la segmentación por instancias. Ambas abordan la tarea de comprender el contenido visual de una imagen, pero se diferencian en sus objetivos y enfoques.

## 2.1 Estado del arte en segmentación semántica

La segmentación semántica tiene como objetivo principal clasificar cada píxel de una imagen según la clase de objeto a la que pertenece. Dada la complejidad del problema, la segmentación semántica de imágenes es una rama que lleva más de una década de mejoras las cuales han centrado sus esfuerzos recientes principalmente en el aprendizaje profundo, particularmente en el desarrollo de arquitecturas especializadas en tareas visuales complejas. A lo largo de la última década, se han propuesto diversos enfoques que difieren en la forma de representar y procesar las características de una imagen, evolucionando desde arquitecturas simples basadas en convoluciones hasta sofisticadas redes con mecanismos de atención y *transformers* [5, 6].

### 2.1.1 Arquitecturas totalmente convolucionales

El punto de partida en el ámbito de la segmentación semántica basada en aprendizaje profundo puede establecerse en el trabajo de Long et al. [7], quienes introdujeron las *Fully Convolutional Networks* (FCNs) en el 2014. Estas redes reemplazaron las tradicionales capas totalmente conectadas (*fully connected*) por convoluciones, permitiendo procesar imágenes con diversas diferencias y con independencia de su tamaño. Este cambio estructural marcó una revolución en la forma de abordar tareas de segmentación, ya que permitió que la red pudiera producir mapas de segmentación con la misma resolución de la imagen de entrada logrando así segmentaciones decentes.

La principal contribución de las FCN fue demostrar que era posible lograr segmentación *end-to-end* usando redes convolucionales profundas, sin etapas intermedias o postprocesamientos complejos. Sin embargo, aunque pioneras, las FCN sufrían problemas relacionados con la pérdida de resolución durante el proceso de *downsampling* debido a su simplicidad, lo cual afectaba la precisión en los bordes y detalles finos. Esto motivó al desarrollo de nuevas arquitecturas que pudieran preservar mejor la estructura espacial de las imágenes.

### 2.1.2 Modelos basados en codificador-decodificador

El paradigma codificador-decodificador se introduce como una solución al problema de pérdida de información espacial presente en las FCN. En este esquema, una red actúa como codificador para extraer características de la imagen de entrada, y otra red, normalmente simétrica, actúa como decodificador para reconstruir la resolución original de la imagen mientras produce el mapa de segmentación.

El modelo propuesto por el equipo de investigadores Badrinarayanan et al. en 2017, SegNet [8], es uno de los primeros en seguir este enfoque. Utiliza una estructura cuyo codificador se basa en VGG16 y un decodificador que emplea los índices de *max pooling* calculados en la etapa de codificación para una reconstrucción más precisa sin requerir aprendizaje adicional en las capas del decodificador.

Otro de los modelos más destacados en este enfoque y que se desarrolla en este trabajo de fin de máster es el propuesto por Ronneberger et al. en 2015, U-Net [9], un avance crucial dentro de esta categoría, diseñado originalmente para

tareas de segmentación en imágenes biomédicas. Su principal innovación fue la introducción de *skip connections* efectivas entre las capas correspondientes del codificador y del decodificador. Estas conexiones directas permiten transferir detalles espaciales finos desde la fase de codificación hasta la de reconstrucción, mejorando significativamente la precisión en regiones pequeñas o con bordes complejos.

Posteriormente, surgieron variantes más sofisticadas como UNet++ propuesta por Zhou et al. en 2019 [10], que refina aún más la incorporación de nodos intermedios y caminos de conexión densamente entrelazados entre múltiples niveles de codificación y decodificación. Permitiendo así un procesamiento más progresivo de las características y facilitando la agregación de información en diferentes niveles para una segmentación más precisa.

### 2.1.3 Arquitecturas basadas en pirámides

La necesidad de capturar información contextual de las imágenes a múltiples escalas dio origen a las arquitecturas piramidales, que buscan incorporar tanto características locales como globales. Este enfoque ha demostrado ser crucial para lograr segmentaciones más coherentes y con mayor contexto semántico, especialmente en escenas complejas con objetos de distintos tamaños.

En este aspecto el modelo *Pyramid Scene Parsing Network* (PSPNet) desarrollado de la mano de Zhao et al. en 2017 [11] fue una de las primeras propuestas en este ámbito. Este modelo introduce el módulo *Pyramid Pooling Module* (PPM), que realiza *pooling* a distintas escalas (de ahí lo del enfoque piramidal) y luego concatena la información para generar un mapa de características enriquecido que integra información contextual global y local del que predice los píxeles para la segmentación.

Por su parte, la familia DeepLab [12-15] una serie de modelos propuestos por Chen et al. entre los años 2014 y 2018, introdujo importantes innovaciones, como las convoluciones dilatadas (*atrous convolutions*), que permiten aumentar el campo receptivo sin perder resolución o el módulo *Atrous Spatial Pyramid Pooling* (ASPP) incorporado a partir de la versión 2, que utiliza múltiples tasas de dilatación para capturar contexto en diversas escalas de manera simultánea. Con el tiempo, versiones mejoradas como DeepLabv3 y DeepLabv3+ introdujeron mecanismos de refuerzo en la decodificación y un mejor manejo de los bordes, consolidándose como una de las arquitecturas más utilizadas y adaptadas en distintos problemas de segmentación semántica.

### 2.1.4 Modelos basados en atención y *transformers*

A medida que se hicieron evidentes las limitaciones de las convoluciones, especialmente en la captura de dependencias a gran escala, surgieron modelos que aprovecharon los mecanismos de atención para superar estos retos. Inicialmente, estos modelos incorporaban técnicas de atención espacial y de canal para mejorar la integración de información contextual, y con el tiempo evolucionaron hacia estructuras completas basadas en *transformers*, lo que les permitió modelar de manera más efectiva relaciones globales en las imágenes.

Entre estos modelos destaca *Dual Attention Network* (DANet), presentado por Fu et al. en 2018 [16], que incorpora un mecanismo de atención dual. Este enfoque consiste en utilizar de manera independiente dos módulos de atención: uno enfocado en las relaciones espaciales y otro en las relaciones de canal, cuyas salidas se acaban fusionando. El módulo de atención espacial (PAM o *Position Attention Module*) se encarga de capturar dependencias globales entre todos los píxeles de la imagen, calculando similitudes entre cada par de posiciones para generar un mapa de atención. Este mapa permite que cada píxel

se reajuste como una combinación ponderada de todos los demás, lo que refuerza la consistencia intraclase y corrige posibles errores derivados de un contexto demasiado local. De este modo, el PAM integra de forma adaptativa la información global, beneficiando especialmente la segmentación de objetos fragmentados o con partes distantes entre sí. Por otro lado, el módulo de atención de canal (CAM o *Channel Attention Module*) explota las interdependencias entre los diferentes canales de las características. CAM calcula un mapa de atención a partir de la multiplicación matricial entre los vectores de características de cada canal. Este mapa permite recalibrar la activación de cada canal en función de su relación con los demás, enfatizando aquellas señales relevantes para la tarea de segmentación y suprimiendo la información redundante.

La llegada de los *transformers* al campo de la visión, inspirados en su éxito en el procesamiento de lenguaje natural, abrió la puerta a innovaciones significativas en la forma de abordar la segmentación semántica. Un ejemplo destacado es SETR (*Segmenter Transformer*) presentado por Zheng et al. en 2020 [17], que es una arquitectura híbrida que combina un codificador-*transformer* y un decodificador convolucional para realizar la segmentación semántica. Su enfoque divide la imagen de entrada en parches (regiones no superpuestas de tamaño fijo), los cuales se convierten en *embeddings* mediante una capa lineal o convolución. Estos *embeddings* se enriquecen con *position embeddings* para preservar la información espacial y se procesan mediante capas de autoatención multicabeza en el codificador-*transformer*, capturando así relaciones de largo alcance entre parches y modelando contexto global. El decodificador, en cambio, utiliza capas convolucionales para aumentar la resolución de los mapas de características y fusionar detalles locales, generando un mapa de segmentación preciso. SETR propone dos variantes: SETR-PUP, que aplica *upsampling progresivo* para recuperar detalles finos, y SETR-MLA, que agrega características multinivel para mejorar la coherencia espacial.

Otro ejemplo en este área es SegFormer presentado en 2021 por Xie et al. [18] que combina un codificador jerárquico basado en *transformers* (*Mix Transformer encoders* o MiT) con un decodificador *All*-MLP ultraligero. El codificador divide la imagen en parches superpuestos y utiliza *Efficient Self-Attention* que aplica un radio de reducción $R$, lo que reduce la complejidad computacional. Además, integra convoluciones dentro del bloque *feed-forward* (Mix-FFN) para añadir información posicional de manera dinámica. El decodificador fusiona las características obtenidas a múltiples escalas mediante capas MLP, lo que aprovecha el amplio campo receptivo del codificador para capturar simultáneamente el contexto global y local.

Mientras que el modelo *Segmenter Transformer* mencionado utiliza un decodificador basado en convoluciones para transformar la secuencia de *embeddings* producida por su codificador-*transformer* en el mapa de segmentación final, el modelo Segmenter, presentado por Strudel et al. en 2021 [19], adopta un enfoque alternativo. En Segmenter, la imagen es dividida en parches pequeños, los cuales se procesan en un codificador basado en *transformers* para generar representaciones jerárquicas y ricas en contexto (similar al modelo anterior), y se diferencia en que su decodificador es ultraligero y se construye exclusivamente con capas MLP, eliminando la necesidad de módulos convolucionales complejos lo que logra mejores resultados.

En cuanto a modelos más recientes, el de Zhou et al., FAN-Hybrid de 2022 [20] y el de Guo et al., Fan-Hybrid-TAP-ADL en 2023 [21] profundizan en el paradigma de redes basadas en atención. FAN-Hybrid propone una arquitectura

mixta que combina las fortalezas de las convoluciones (a través de bloques ConvNeXt en las etapas inferiores) con bloques *Fully Attentional Network* (FAN) basados en atención global en las etapas superiores. Esta combinación permite capturar tanto patrones visuales locales como dependencias contextuales de largo alcance, logrando así una mayor precisión y robustez frente a perturbaciones en los datos. Por su parte, FAN-Hybrid-TAP-ADL aborda la fragilidad de los *transformers* frente a datos corruptos mediante dos innovaciones: la primera es *Token-aware Average Pooling* (TAP), que ajusta dinámicamente el área de influencia de cada token (representaciones vectoriales de pequeñas regiones de una imagen) utilizando un *pooling* con dilatación variable, incorporando información de los píxeles vecinos para reducir posible ruido; y la segunda *Attention Diversification Loss* (ADL), una función de pérdida que penaliza la similitud entre vectores de atención, promoviendo patrones de atención más diversos para evitar el "sobreenfoque" en pocos tokens.

### 2.1.5 Modelos más recientes

El progreso reciente en segmentación semántica se ha caracterizado por el diseño de modelos híbridos que combinan lo mejor de cada enfoque anterior. Estas arquitecturas buscan un equilibrio entre precisión, eficiencia computacional y capacidad de generalización. Entre ellas se destacan propuestas como InternImage [22] y SERNet-Former [23].

InternImage es un modelo desarrollado por Wang et al. cuya última aproximación se publica en 2023 cuya base son las redes neuronales convolucionales (CNN) y está diseñado para escalar eficientemente a grandes volúmenes de datos (hasta 400 millones de imágenes) y parámetros (hasta 1000 millones). Su innovación principal es el uso de convoluciones deformables v3 (DCNv3) , que permiten campos receptivos adaptativos y dependencias de largo alcance mediante desplazamientos dinámicos de muestreo. Además, integra componentes inspirados en *transformers*, como *layer normalization* (LN) y redes *feed-forward* (FCN), para mejorar su capacidad de modelado.

Por otro lado, SERNet-Former presentado en 2024 por Eris se orienta hacia la segmentación semántica mediante una arquitectura codificador-decodificador optimizada. Su codificador, basado en una variante eficiente de redes residuales enriquecidas con mecanismos de atención (a través de *attention-boosting gates* y módulos), mejora la extracción de características relevantes sin aumentar excesivamente la carga computacional. El decodificador, complementado con redes de fusión de atención, refina y combina la información multiescala para lograr una segmentación precisa.

## 2.2 Estado del arte en segmentación por instancia

En el caso de la segmentación por instancias el objetivo final difiere del de la semántica, aquí se busca identificar y delimitar cada objeto presente en una imagen, asignándole una identificación única. Este enfoque implica no solo clasificar cada píxel según la categoría a la que pertenece, sino también diferenciar entre distintos objetos de la misma clase. La evolución en este campo ha estado marcada por la diversidad de enfoques, que han evolucionado desde modelos pioneros basados en propuestas hasta métodos que abordan el problema de forma directa y, que recientemente, incorporan *transformers* para capturar relaciones de largo alcance [24, 25].

### 2.2.1 Enfoques basados en propuestas

Los métodos basados en propuestas han sido fundamentales en el desarrollo de la segmentación por instancias. Estos parten de la idea de generar regiones

candidatas que posteriormente se refinan para obtener máscaras precisas de cada objeto.

Entre los primeros enfoques en esta línea se encuentra el de *Instance-Sensitive Fully Convolutional Network*, un modelo presentado por Dai et al. en el año 2016 [26]. Aunque las FCN originalmente se diseñaron para la segmentación semántica, posteriormente se adaptaron para generar mapas enfocados a instancias. Esto se consigue mediante la utilización de dos ramas: una que extrae características de la imagen para producir mapas de puntuación a nivel de píxel, y otra que, mediante un módulo de ensamblaje que emplea una técnica de ventana deslizante, utiliza dichos mapas para localizar y delimitar individualmente cada objeto.

Otro avance en esta categoría fue la introducción del modelo presentado por He et al. en 2017, Mask R-CNN [27]. Inspirado por Faster R-CNN, Mask R-CNN incorpora una rama adicional para la predicción de máscaras. Esta máscara se genera en paralelo a las ramas de clasificación y regresión de su modelo predecesor, a partir de las regiones de interés (ROI) que se extraen del mapa de características de una red CNN. Este diseño modular y flexible permite obtener máscaras detalladas para cada objeto, integrando de forma simultánea la detección y la segmentación.

Otro enfoque es el de *Path Aggregation Network* (PANet) un modelo desarrollado por Liu et al. en 2018 [28], el cual se propone mejorar la transmisión de información desde las capas de bajo nivel hasta las de alto nivel. PANet refina la arquitectura de Mask R-CNN al introducir una técnica de aumento *bottom-up* que acorta la distancia entre las características de baja y alta resolución. Además, se emplea un *pooling* de características adaptativo para recuperar la ruta de información fragmentada entre todos los niveles de características y para cada propuesta. Finalmente, PANet incorpora una rama adicional basada en capas totalmente conectadas que captura la información desde una perspectiva alternativa, facilitando la integración de detalles finos en la segmentación final.

La *Hybrid Task Cascade* (HTC) surge como una propuesta de la mano de Chen et al. en 2019 [29] que combina en un único marco los conceptos de cascada de refinamiento y la generación de máscaras. HTC fusiona de forma conjunta las tareas de refinamiento en múltiples etapas, permitiendo que las mejoras obtenidas en cada fase se retroalimenten para potenciar la calidad de la segmentación. Asimismo, incorpora una rama completamente convolucional en la que se provee del contexto espacial necesario para diferenciar eficazmente el primer plano del fondo, permitiendo una integración gradual y selectiva de las características a lo largo del proceso.

Finalmente, *You Look Only At CoeffcienTs* (YOLACT) de Bolya et al. presentado en 2019 [30] muestra una solución orientada a la segmentación por instancias en tiempo real. Este método adopta un enfoque de detector de una sola etapa, dividiendo el proceso en dos módulos concurrentes. El primero consiste en una rama que, mediante una red FCN, genera un conjunto de máscaras prototipo (propuestas) a partir de la imagen completa. Paralelamente, una segunda rama predice un vector de coeficientes para cada propuesta, que actúa como un codificador de la representación de la instancia. La combinación lineal de estos coeficientes permite reconstruir la máscara final para cada objeto, logrando una segmentación rápida y eficiente sin necesidad de etapas complejas de refinamiento.

### 2.2.2 Enfoques libres de propuestas

Los métodos libres de propuestas abordan la segmentación por instancias sin recurrir a una etapa preliminar de generación de regiones, enfocándose en segmentar directamente los objetos a partir de técnicas de *pooling* o codificación a nivel de píxel. La idea central es aprovechar la capacidad de la segmentación semántica para obtener mapas de características robustos y, a partir de ellos, agrupar o clusterizar píxeles que pertenezcan a la misma instancia. A continuación se describen algunos de los métodos más representativos en esta línea:

*Deep Watershed Transform* es una aproximación para segmentación por instancia presentada por Bai y Urtasun en 2016 [31] que combina la tradicional transformación *watershed* con técnicas de aprendizaje profundo para generar un mapa de energía en el que las depresiones (o "cuencas") representan las diferentes instancias. La red aprende a producir un mapa energético en el que cada cuenca corresponde a un objeto, y se emplea una técnica de corte a un único nivel de energía para evitar la sobresegmentación. De este modo, la red segmenta de forma directa las instancias mediante un proceso *end-to-end*, simplificando el *pipeline* al evitar etapas de generación de propuestas.

En 2019 surge RetinaMask de la mano de Fu et al. [32]como una extensión del detector de una sola etapa RetinaNet, integrando una rama adicional para la predicción de máscaras. La idea es que, durante el entrenamiento, se optimice la red no solo para la detección mediante clasificación y regresión de *bounding* boxes (como en RetinaNet), sino también para la predicción de máscaras de instancias. Esto se consigue mediante la incorporación de una nueva rama para máscaras, junto con la adaptación de la función de pérdida. Además, se aplican técnicas como la normalización por grupos que mejora la estabilidad del entrenamiento, manteniendo el coste computacional similar al de RetinaNet y permitiendo obtener máscaras de alta calidad de manera directa.

PolarMask propuesto por Xie et al. en 2019 [33] consiste en un enfoque anclado en la representación polar para la segmentación por instancias. En lugar de predecir directamente *bounding boxes* o máscaras en el espacio cartesiano, el método transforma la representación de la instancia al espacio polar, generalizando detectores de cajas como *Fully Convolutional One-Stage Object Detection* (FCOS) [34] al modelar máscaras con contornos polares. La red predice la forma del contorno mediante dos tareas paralelas basadas en representación polar: clasificación del centro del objeto y regresión densa de distancias en ángulos discretos desde ese centro hacia el contorno. Esto permite que el ángulo sea naturalmente direccional, lo que hace más conveniente conectar los puntos en un contorno entero. Se implementan técnicas como el Polar IoU *Loss* (para optimizar la superposición de máscaras) y el Polar *Centerness* (para seleccionar centros de alta calidad), integrando todo en una arquitectura de una sola etapa sin dependencia de propuestas regionales o *pipelines* multifase. Esto simplifica significativamente el flujo de trabajo frente a métodos tradicionales.

El modelo SOLO (*Segmenting Objects by Locations*) de Wang et al. presentado en 2019 [35] plantea la segmentación por instancias como un problema de predicción densa, donde cada celda de una cuadrícula SxS se asigna a una instancia (si el centro del objeto está dentro de ella), prediciendo su categoría semántica y generando una máscara en coordenadas normalizadas. En lugar de la estrategia 'detectar y segmentar', SOLO utiliza convoluciones sensibles a la posición (CoordConv) para segmentar objetos directamente en una sola etapa, asociando cada máscara a una celda específica de la cuadrícula.

SOLOv2 [36] sale un año más tarde de la mano de los mismos autores mejorando este enfoque al introducir una metodología dinámica para segmentar las ubicaciones de los objetos. Se incorporan mecanismos de aprendizaje de *kernels* convolucionales dinámicos y de características, y se emplea la técnica de *matrix Non-Maximum Suppression* (NMS), que actúa de forma paralela para suprimir predicciones duplicadas y afinar la calidad de la máscara final. Esta evolución optimiza tanto la eficiencia como la precisión del sistema, permitiendo una segmentación por instancias en tiempo real con una arquitectura más simplificada.

### 2.2.3 Modelos basados en *transformers*

Con respecto a los *transformers* en segmentación por instancia, estos aplican una filosofía diferente en diseño respecto a los enfoques presentados anteriormente. Dicha filosofía se muestra a continuación:

El enfoque ISTR (*End-to-End Instance Segmentation with Transformers*) se muestra en el artículo de Hu et al. del año 2021 [37] y se centra en eliminar componentes tradicionales como NMS mediante el uso de emparejamiento bipartito, lo que ayuda a reducir resultados redundantes en la predicción. A diferencia de los modelos convencionales, ISTR propone una segmentación de instancias integral con *transformers* que se enfoca en predecir *embeddings* de máscara a nivel bajo, facilitando el entrenamiento incluso con conjuntos de datos pequeños. Además, incorpora una estrategia de refinamiento recurrente que opera en paralelo para detectar y segmentar los objetos, integrando de forma efectiva tanto la detección como la segmentación dentro de un marco *end-to-end*.

El equipo Guo et al. en 2021 presentan SOTR (*Segmenting Objects with Transformers*) [38]. Este modelo utiliza la capacidad de los *transformers* para abordar problemas como el *clustering* inestable o sobreajuste mediante el aprendizaje de características sensibles a la ubicación, permitiendo extraer características globales y contextuales. SOTR, además, introduce un mecanismo de doble atención (*twin attention mechanism*) que reduce el coste computacional y de memoria, y permite que la red combine las ventajas de las arquitecturas basadas en CNN y *transformers*, logrando una mejor segmentación sin depender de técnicas de *pooling* convencionales.

El modelo SOIT (*Segmenting Objects with Instance-Aware Transformers*) presentado por Yu et al. en 2021 [39], redefine la segmentación por instancias como un problema de predicción de conjuntos, eliminando la necesidad de componentes manuales como el recorte de regiones de interés (RoI) y de postprocesamiento con NMS. En lugar de ello, SOIT aprende consultas (*queries*) que codifican simultáneamente la categoría, la posición y la máscara píxel a píxel de cada objeto (son vectores que actúan como "preguntas aprendidas" que se utilizan para buscar y capturar información específica de un objeto). Este enfoque permite un entrenamiento *end-to-end* en el que cada instancia se segmenta de forma directa y simultánea, facilitando la integración de la detección y la segmentación en un único proceso. La capacidad de SOIT para manejar múltiples tareas de forma concurrente lo hace especialmente útil para aplicaciones en las que se requiere una integración estrecha entre la detección y la segmentación.

Por último, Mask2Former es un modelo universal de segmentación propuesto por Cheng en 2021 [40] basado en transformers que unifica tareas semánticas y por instancias a través de *set prediction* (predicción de conjuntos). Utiliza atención enmascarada para enfocarse solo en regiones relevantes de cada

máscara. Procesa características multiescala (pirámide de características) de forma cíclica para detectar objetos pequeños y grandes eficientemente, y utiliza un *transformer* en el decodificador. Elimina pasos manuales como NMS gracias al *set prediction* y logra alta precisión en una sola etapa.

### 2.2.4 Modelos más recientes

En cuanto a los modelos más recientes que marcan un avance importante en segmentación por instancia se encuentran los siguientes:

QueryInst desarrollado por Fang en 2021 [41], es otro modelo basado en consultas (de forma similar a SOIT) que utiliza cabezales dinámicos de máscara (DynConvmask) con supervisión paralela en seis etapas para extraer información relevante de objetos. Su innovación radica en aprovechar la correspondencia uno a uno entre consultas en diferentes etapas y entre características de máscara (mask RoI) y consultas en la misma etapa, eliminando la necesidad de propuestas predefinidas (regiones de interés). Durante el entrenamiento, cada consulta actúa como memoria dinámica: los módulos DynConvmask adaptan las características locales mediante convoluciones guiadas por las consultas, permitiendo un flujo de información por máscara que integra contexto local (FPN) y global (autoatención). En inferencia, solo se usa la etapa final, donde las consultas consolidan información multietapa para generar máscaras precisas. Además, QueryInst comparte consultas y mecanismos de autoatención (MSA o *Multi-Head Self-Attention*) entre las tareas de detección y segmentación, sincronizando ambas y evitando inconsistencias en la distribución de propuestas.

El modelo SparseInst se publica en el año 2022 por Cheng et al. [42], que utiliza un conjunto escaso de *instance activation maps* para resaltar de manera directa las regiones útiles de cada objeto sin necesidad de propuestas predefinidas ni procesos de NMS. Su arquitectura integral se compone de un *backbone* para extraer características multiescala, un codificador de contexto de instancia que fusiona información de diversas resoluciones mediante *pooling* piramidal y un decodificador *instance activation maps* (IAM) dividido en dos ramas: una para generar los mapas de activación y extraer vectores compactos y discriminativos para el reconocimiento, y otra para codificar las características de máscara. Durante el entrenamiento, la red aprende a destacar regiones relevantes a través de supervisiones indirectas proporcionadas por módulos posteriores de clasificación y segmentación, utilizando asignación de etiquetas vía *matching* bipartito y un ajuste de *objectness* basado en IoU. Esta integración de componentes y estrategias permite a SparseInst consolidar la extracción y utilización de representaciones visuales de manera veloz y precisa en la detección y segmentación a nivel de instancia.

SAM (*Segment Anything Model*) es un modelo de Meta diseñado por Kirillov en 2023 [43], con el objetivo de generar máscaras a partir de diversos tipos de *prompts* sin restricciones, combinando un codificador de imagen basado en *vision transformer* preentrenado con *Masked Autoencoder* (MAE) para procesar imágenes de alta resolución, un *prompt* codificador flexible que integra distintas entradas como puntos, cajas o texto y máscaras *dense*, y un decodificador eficiente de máscaras que utiliza bloques *transformer* modificados con atención bidireccional para actualizar las características. Este modelo resuelve ambigüedades generando múltiples máscaras para un único *prompt*, optimizando la precisión mediante puntuaciones de confianza basadas en IoU.

Los modelos YOLOv8 de Reis et al. en 2023 [44], YOLOv9 de Wang et al. en 2024 [45] y YOLOv10 de Wang et al. en 2024 [46] son las versiones más recientes de

la familia YOLO, ampliamente reconocida por su eficiencia en tareas de detección de objetos en tiempo real. Estas versiones han evolucionado significativamente, no solo en velocidad y precisión, sino también en su capacidad para abordar tareas más complejas como la segmentación por instancias.

A partir de YOLOv8, se introducen cabezas de segmentación adicionales, permitiendo realizar segmentación de instancias en un solo paso mediante mínimas modificaciones a la arquitectura base de detección. Esto combina la velocidad de los detectores de una sola etapa con una mayor precisión en la generación de máscaras.

YOLOv9 amplia esta capacidad introduciendo innovaciones como *Programmable Gradient Information* (PGI), una técnica diseñada para preservar la información completa de la entrada durante el entrenamiento, y la arquitectura *Generalized Efficient Layer Aggregation Network* (GELAN), diseñada para reducir la pérdida de información y mejorar la eficiencia en el entrenamiento.

YOLOv10 da un paso más al eliminar completamente la necesidad de NMS mediante un sistema de entrenamiento e inferencia sin NMS basado en dos componentes: *Dual Label Assignments* que utiliza dos cabezas durante el entrenamiento. La cabeza uno-a-muchos que proporciona una supervisión densa al asignar múltiples predicciones a cada instancia, y la cabeza uno-a-uno que asegura una correspondencia estricta entre predicciones y verdades absolutas. Durante la inferencia, solo se utiliza la cabeza uno-a-uno, eliminando la necesidad de NMS y reduciendo la latencia. El otro componente es *Consistent Matching Metric* que alinea los objetivos de entrenamiento de ambas cabezas utilizando una métrica uniforme que evalúa la concordancia entre predicciones e instancias. Esto garantiza que las mejores predicciones seleccionadas por la supervisión uno-a-muchos también sean válidas para la cabeza uno-a-uno, mejorando el rendimiento general del modelo. Con esto se consigue optimizar la precisión y reducir la latencia.

Grounded-SAM aparece en 2024 de la mano de Ren [47] combinando la detección con vocabulario abierto de Grounding DINO y con la segmentación precisa de SAM, permitiendo segmentar cualquier objeto mencionado en un *prompt* de texto, sin necesidad de reentrenamiento. Grounding DINO localiza los objetos mediante descripciones textuales, y SAM genera las máscaras a partir de esas ubicaciones. Esta integración facilita tareas de segmentación abierta incluso en categorías raras, y habilita aplicaciones como anotación automática, edición de imágenes guiada por texto y análisis de movimiento humano.

# 3. Fundamentos teóricos

En este capítulo se abordan los fundamentos teóricos que sustentan las técnicas avanzadas de segmentación en imágenes que se van a implementar. Se examinan en detalle dos enfoques distintos: por un lado, los modelos dedicados a la segmentación semántica: U-Net y DeepLab v3+; y por otro, los modelos orientados a la segmentación por instancia, ejemplificados por Mask R-CNN.

## 3.1 Modelos empleados para segmentación Semántica

A continuación, se profundiza en los modelos destinados a la segmentación semántica, donde cada píxel de la imagen se clasifica según la categoría a la que pertenece. Se presentan y analizan las arquitecturas de U-Net y DeepLab v3+, dos enfoques que han demostrado gran eficacia en tareas de segmentación en el ámbito médico y en otros sectores de visión por ordenador. La discusión abarca los fundamentos teóricos que sustentan su funcionamiento práctico y matemático, así como sus arquitecturas convolucionales permitiendo comprender cómo se logra una segmentación precisa a nivel de píxel.

### 3.1.1 U-Net

U-Net es una arquitectura de red neuronal convolucional diseñada específicamente para tareas de segmentación semántica de imágenes, que fue introducida en 2015 por Olaf Ronneberger, Philipp Fischer y Thomas Brox [9].

U-Net surgió ante la necesidad de segmentar imágenes biomédicas con alta precisión. Antes de su aparición, muchas arquitecturas de redes neuronales no lograban capturar con exactitud los contornos y las estructuras finas de células o tejidos, ya que la información espacial se perdía progresivamente a lo largo de las capas. U-Net, denominada así por su distintiva forma en "U", se inspira en la estructura de los autocodificadores [48], que son redes neuronales diseñadas para aprender representaciones densas y compactas de los datos mediante aprendizaje no supervisado. En un autocodificador típico, la arquitectura se asemeja a un reloj de arena, donde los datos se comprimen en el codificador hasta alcanzar lo que se denomina representación latente y luego se reconstruyen en la parte de decodificación, de modo que el número de neuronas de salida coincide con el número de entradas. A continuación se muestra un ejemplo de autocodificador convencional:

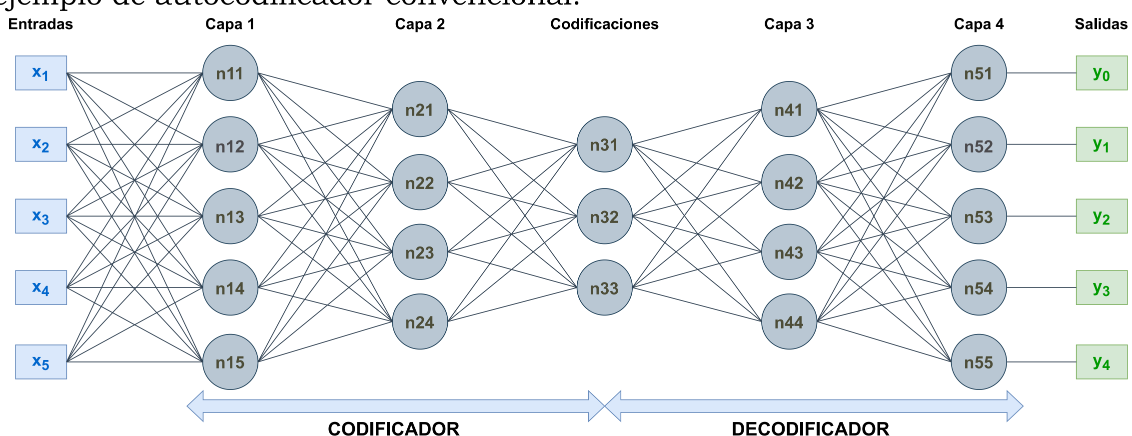



*Figura 1. Arquitectura del autocodificador*

No obstante, en el caso de U-Net a diferencia de los autocodificadores tradicionales, como el presentado, basados en MLP (*MultiLayer Perceptron*), se utilizan operaciones convolucionales para preservar y recuperar la información espacial, ya que estas convoluciones son una buena aproximación para extraer

características de las imágenes, lo que resulta muy útil en tareas de segmentación.

Con respecto a su entrenamiento la red U-Net recibe imágenes junto con sus máscaras de segmentación correspondientes, donde cada píxel está etiquetado según la clase a la que pertenece (en segmentación multiclase), y devuelve un vector de probabilidades con K clases por cada píxel mediante la función de activación Softmax.

La red se entrena utilizando como función de pérdida la entropía cruzada ponderada por píxeles (*pixel-wise weighted cross-entropy*), que compara la clase asignada a cada píxel según la red, con su clase real.

$$L = \frac{1}{N} \sum_{X \in \Omega} w(x) \sum_{C=1}^{C} y_c(c) \log p_c(x),$$

donde Ω es el conjunto de píxeles de la imagen, N el total de píxeles, C el número total de clases, $y_c(x)$ la etiqueta verdadera en formato *one-hot* para la clase c en el píxel x ($y_c(x)=1$ si el píxel x pertenece a la clase c, 0 en otro caso), $p_c(x)$ es la probabilidad predicha de que el píxel x pertenezca a la clase c y w(x) es el mapa de pesos precalculado para enfatizar los píxeles más importantes. Esto último sirve para compensar la frecuencia de píxeles de cada clase y asignar mayor peso a los píxeles cercanos a los bordes entre objetos (separa instancias adyacentes), la ecuación del mapa de pesos se describe en el artículo original de la siguiente forma:

$$w(x) = w_c(x) + w_0 * \exp\left(-\frac{(d_1(x) + d_2(x))^2}{2\sigma^2}\right),$$

donde $w_c(x)$ es el peso para balancear clases (mayor para clases poco comunes), $d_1(x)$ y $d_2(x)$ para distancias al primer y segundo borde más cercano y $w_0$ y σ hiperparámetros (establecidos en el artículo a 10 y 5 respectivamente).

A continuación, se muestra una figura en la que se presenta la arquitectura U-Net:

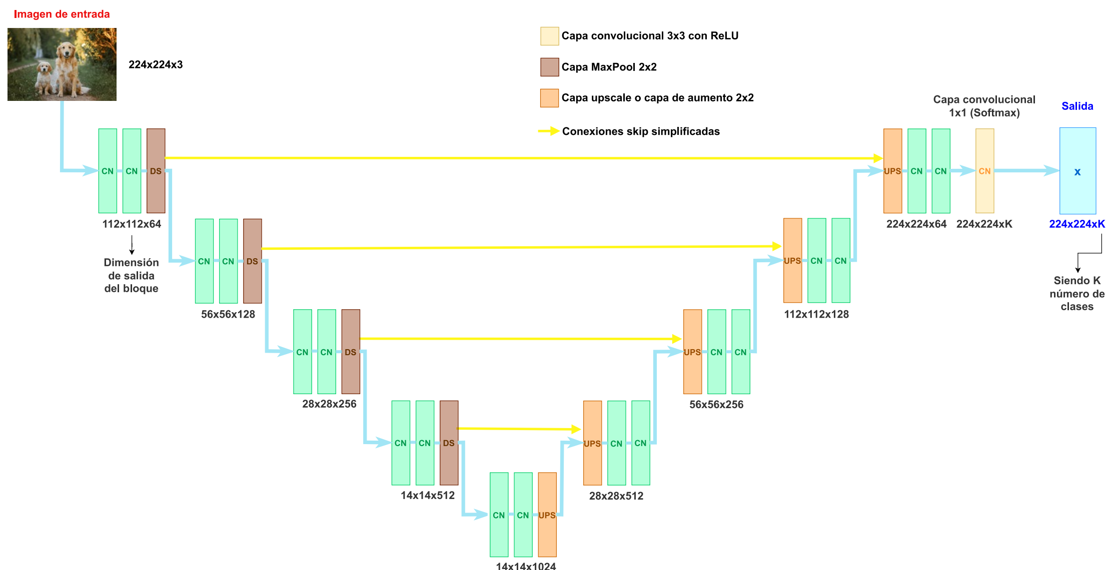



*Figura 2. Arquitectura de la U-Net*

Como se observa en la figura la arquitectura de U-Net se divide en dos ramas principales:

- **Codificador:** En esta parte, la red aplica operaciones de convolución seguidas de funciones de activación (ReLU) y de *max pooling*. Cada bloque del codificador reduce la resolución espacial de la imagen, lo que permite extraer características de alto nivel y capturar el contexto global.
- **Decodificador:** Para recuperar la información espacial y lograr una segmentación precisa, el decodificador emplea técnicas de *upsampling* (por ejemplo, convoluciones transpuestas) que incrementan el tamaño de los mapas de características. Además, en cada etapa del decodificador se establecen *skip connections* que conectan las salidas correspondientes del codificador con sus entradas. Estas conexiones permiten que el decodificador reciba información de alta resolución que se perdió durante el *pooling*, facilitando la reconstrucción detallada de los contornos y estructuras pequeñas.

## 3.1.2 DeepLabV3+

### 3.1.2.1 DeepLab v1

La familia de modelos DeepLab surge en diciembre de 2014 con el lanzamiento de DeepLabv1, desarrollado por el equipo de Google Research y dirigido por Liang-Chieh Chen [12], con el objetivo de mejorar la segmentación semántica de imágenes.

Aunque DeepLab v1 aparece prácticamente al mismo tiempo que U-Net, aborda el problema de la segmentación de manera diferente. En lugar de utilizar una arquitectura basada en autocodificadores, DeepLabv1 se construye a partir de una versión modificada de la red VGG16 [49]. La arquitectura de VGG16 se muestra a continuación:

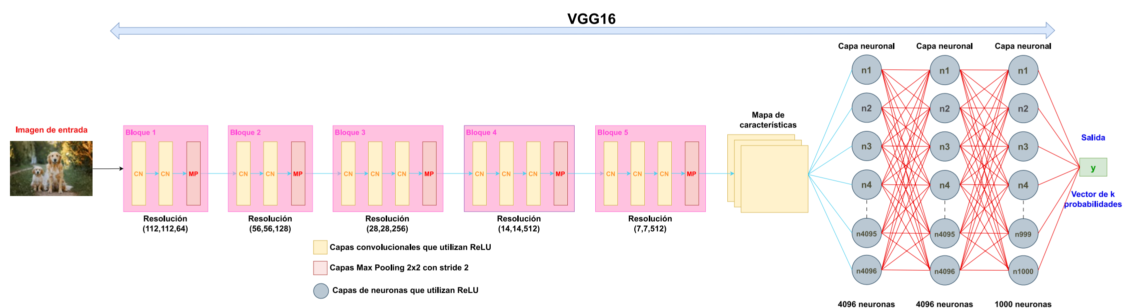



*Figura 3. Arquitectura de VGG16*

DeepLab v1 modifica concretamente el bloque 5 de la red VGG16 eliminando la capa *max pooling* que ya no se necesita debido a un cambio en las convoluciones normales por convoluciones dilatadas (*atrous convolutions*). Estas convoluciones amplían el campo receptivo de los filtros al insertar espacios entre los pesos del *kernel*, controlados por un parámetro llamado tasa de dilatación. De este modo, la red logra capturar un contexto más amplio sin reducir la resolución del mapa de características, preservando mejor los detalles espaciales y evitando la pérdida de información importante en la segmentación. A continuación, en la figura 4 se muestra su funcionamiento:

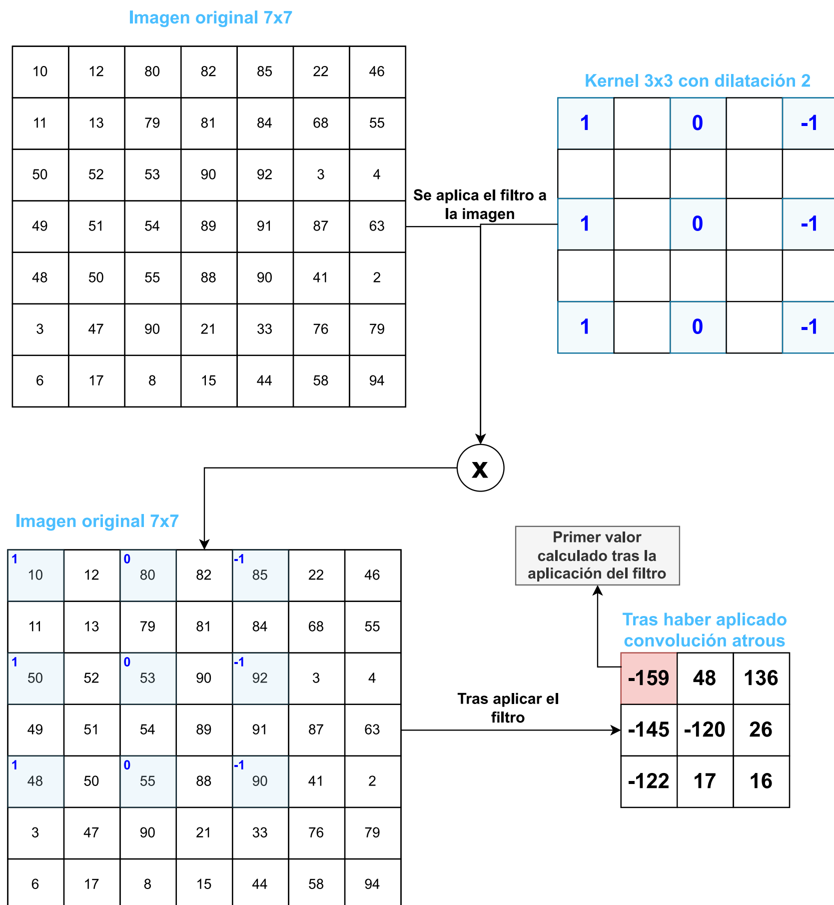



*Figura 4. Esquema del funcionamiento de la convolución atrous*

Otra modificación por parte de DeepLab v1 al modelo VGG16 es el cambio de sus capas Dense por dos capas *atrous* y una convolucional, que permite asignar a cada píxel la probabilidad de pertenecer a cada clase, seguidas de una capa *upsampling* bilineal que realiza interpolación bilineal sobre la salida de esta última capa y así redimensiona la salida de la capa convolucional al tamaño de la imagen original.

La red, por tanto, recibe una imagen, genera un mapa de segmentación y aprende comparando directamente las probabilidades predichas (después del *upsampling*) con la máscara de *ground truth* (los valores reales) usando de función de pérdida la entropía cruzada *pixel-wise*, al igual que en U-Net.

Por último, una vez que la red ha aprendido. DeepLab v1 añade un posprocesamiento en la fase de inferencia mediante CRF (*Conditional Random Field*) denso, que es un modelo probabilístico utilizado en visión por ordenador para refinar la segmentación de imágenes, con el objeto de refinar la máscara de segmentación. Este modelo se basa en la idea de que los píxeles cercanos, y con características similares, tienden a compartir la misma etiqueta de clase, lo que permite corregir errores en la segmentación inicial generada por la red.

El funcionamiento del CRF básico se fundamenta en la definición de una función de energía global que integra dos tipos de términos: los potenciales unarios y los potenciales de pares. La energía total se expresa de la siguiente forma:

$$E(x) = \sum_{i} \theta_i(x_i) + \sum_{ij} \theta_{ij}(x_i, x_j).$$

Aquí, $\theta_i(x_i)$ representa el potencial unario, que mide la compatibilidad de la etiqueta $x_i$ asignada al píxel i según la evidencia local, es decir, las probabilidades que la red ha calculado para cada etiqueta. Este potencial se define como:

$$\theta_i(x_i) = -\log P(x_i),$$

donde $P(x_i)$ es la probabilidad de que el píxel i tenga la etiqueta $x_i$.

Por otro lado, el potencial de pares $\theta_{ij}(x_i, x_j)$ captura las relaciones contextuales entre pares de píxeles i y j. Este término se formula para penalizar asignaciones inconsistentes entre píxeles vecinos, fomentando que píxeles cercanos y con características similares (como color y posición) tengan la misma etiqueta. Su ecuación general es:

$$\theta_{ij}(x_i, x_j) = \mu(x_i, x_j) \sum_{m=1}^{K} w_m k_m(f_i, f_j),$$

donde $\mu(x_i, x_j)$ es la función indicadora del Potts *model* (vale 1 si $x_i \neq x_j$ y 0 si $x_i = x_j$), $k_m(f_i, f_j)$ son *kernels* gaussianos que miden la similitud entre las características $f_i$ y $f_j$ de los píxeles, y $w_m$ son los pesos asignados a cada *kernel*.

En el caso particular de DeepLab v1 se emplean dos *kernels* (es decir, K=2). El primero depende tanto de la posición p como del color I (intensidad) de los píxeles:

$$k_1(f_i, f_j) = \exp\left(-\frac{\|p_i - p_j\|^2}{2\sigma_\alpha^2} - \frac{\|I_i - I_j\|^2}{2\sigma_\beta^2}\right),$$

y el segundo depende únicamente de la posición:

$$k_2(f_i, f_j) = \exp\left(-\frac{\|p_i - p_j\|^2}{2\sigma_\gamma^2}\right).$$

Los parámetros $\sigma_\alpha$, $\sigma_\beta$ y $\sigma_\gamma$ controlan la influencia de cada término, mientras que $w_1$ y $w_2$ determinan la importancia relativa de cada *kernel* en el modelo. Dichos valores son calculados mediante optimizaciones que buscan hacer que la energía total sea mínima. La expresión final de $\theta_{ij}(x_i, x_j)$ una vez presentados los anteriores términos es:

$$\theta_{ij}(x_i, x_j) = \mu(x_i, x_j)\left(w_1 \exp\left(-\frac{\|p_i - p_j\|^2}{2\sigma_\alpha^2} - \frac{\|I_i - I_j\|^2}{2\sigma_\beta^2}\right) + w_2 \exp\left(-\frac{\|p_i - p_j\|^2}{2\sigma_\gamma^2}\right)\right).$$

Lo que se pretende con esta energía es que cada píxel tenga el valor mínimo de energía (este etiquetado de la mejor forma posible). Para la energía global de cada píxel se calcula la penalización que recibe por el resto de los píxeles y se busca minimizar dicha penalización. Un ejemplo del funcionamiento de CRF se puede ver en el apéndice A.1.

Analizar todas las penalizaciones es una tarea computacionalmente intratable, es por ello por lo que se utilizan métodos aproximados como el que mencionan en el artículo de DeepLab v1 que es *Mean Field Approximation*.

La figura 5 describe de manera integral la estructura del modelo DeepLab v1 previamente mencionado:

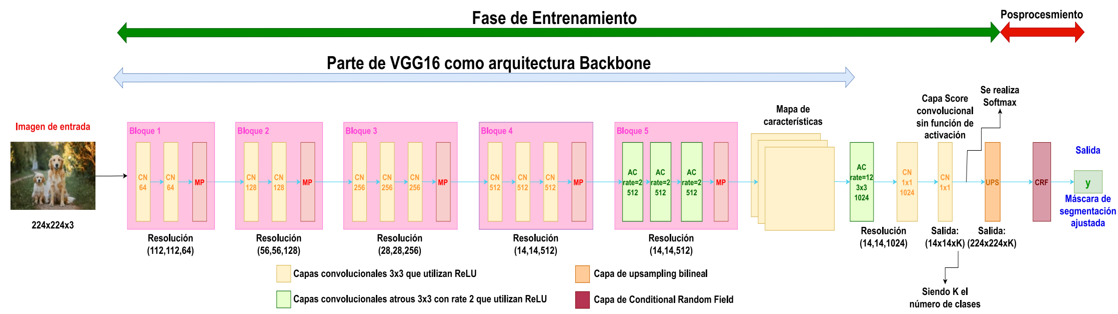



*Figura 5. Arquitectura de DeepLab v1*

### 3.1.2.2 DeepLab v2

En junio de 2016, el equipo de Google Research, liderado nuevamente por Liang-Chieh Chen, presenta la siguiente evolución de la familia DeepLab, DeepLab v2 [13].

DeepLab v2 mejora la capacidad de segmentación de DeepLab v1 al incorporar el módulo *Atrous Spatial Pyramid Pooling* (ASPP). En lugar de emplear una única tasa de dilatación como en v1, ASPP utiliza varias convoluciones *atrous* en paralelo, cada una con una tasa de dilatación diferente (r = 6, 12, 18 y 24, para el caso ASPP-L). Esta estrategia permite que la red capture tanto detalles finos como información global, lo que resulta fundamental para segmentar objetos de diferentes tamaños en una misma imagen. Este módulo se estructura de la siguiente manera:

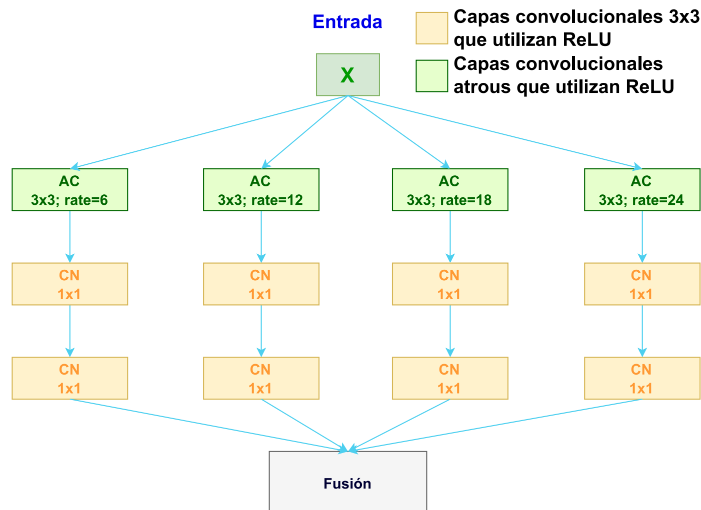



*Figura 6. Esquema del funcionamiento de ASPP*

El módulo ASPP se incorpora justo a la salida del *backbone* de la red, un *backbone* más potente que el de v1, ResNet-101 [50]. ResNet-101, con sus bloques residuales, facilita el entrenamiento de redes muy profundas, lo que se traduce en mejores mapas de características. Con la utilización de convoluciones *atrous* en las últimas capas de ResNet-101, la red mantiene la resolución del mapa de características sin perder la información espacial, al igual que en DeepLab v1, pero con una mayor capacidad de abstracción.

El proceso de segmentación en DeepLab v2 sigue una línea similar a la de v1:

1. La imagen se introduce en la red, la cual extrae un mapa de características de alta resolución a través del *backbone* (ResNet-101) modificado con convoluciones dilatadas (bloque 4 y 5).

2. Sobre este mapa se aplica el módulo ASPP, que genera múltiples mapas de características a diferentes escalas y luego los concatena, formando un mapa enriquecido que contiene información contextual variada.

3. Unas capas de convolución 1x1 reducen la dimensión de los canales para producir un mapa de *logits*, y se aplica interpolación bilineal para remapear el resultado a la resolución original de la imagen.

4. Finalmente, se utiliza un CRF denso en la fase de posprocesamiento para refinar los contornos y ajustar las etiquetas de cada píxel.

A continuación se muestra un esquema que detalla la arquitectura completa de DeepLab v2:

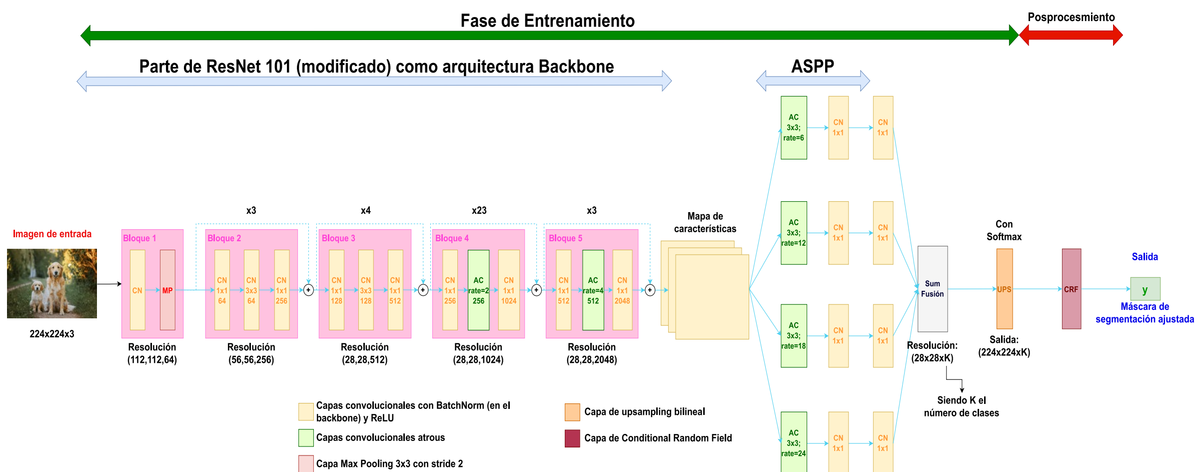


*Figura 7. Arquitectura de DeepLab v2*

### 3.1.2.3 DeepLab v3

DeepLab v3 fue presentada alrededor de 2017 [14] como una mejora sobre DeepLab v2, con el objetivo de capturar información contextual a múltiples escalas de manera más efectiva y de reducir la dependencia en técnicas de posprocesamiento como el CRF. Mientras que DeepLab v1 se basaba en una versión modificada de VGG16 y DeepLab v2 introdujo el módulo *Atrous Spatial Pyramid Pooling* utilizando múltiples tasas de dilatación, DeepLab v3 refina y amplía este concepto.

Una de las innovaciones más destacadas es la incorporación de *Multi-Grid* en la *backbone*, la cual sigue siendo ResNet-101, una estrategia que permite aplicar tasas de dilatación escalonadas dentro de un mismo bloque residual. Mientras que en DeepLab v2 las tasas de dilatación en las capas profundas eran fijas, en DeepLab v3 estas tasas varían jerárquicamente. En DeepLab v3 el *Multi-Grid* se aplica al bloque conv5_x y sus tasas varían en cada bloque residual (de los 3 que tiene) siendo 4 la tasa del primero, 8 la del segundo y 16 las del tercero lo que amplía el campo receptivo de la red sin sacrificar detalles finos.

Otra mejora se encuentra en la evolución del módulo ASPP. En DeepLab v2, ASPP contaba con cuatro ramas, formadas por cuatro convoluciones *atrous* con

distintas tasas de dilatación seguidas de dos capas convolucionales 1x1. DeepLab v3 expande este diseño con una quinta rama basada en *Global Average Pooling* (GAP), que permite capturar información contextual a nivel global. GAP reduce la dimensionalidad espacial de los mapas de características a un solo valor por canal, esto lo hace promediando los valores de todos los píxeles de cada canal para obtener el valor resultante. La salida de GAP se redimensiona y se concatena con las otras ramas de ASPP, igual que se hacía en v2 en la parte de la concatenación, generando una representación más rica que combina detalles locales y contexto general de la imagen.

Además, DeepLab v3 optimiza el flujo de entrenamiento al extender el uso de *Batch Normalization* a todas las capas, incluyendo ASPP y el clasificador. Esto mejora la estabilidad del entrenamiento y permite utilizar mayores tasas de aprendizaje, lo que acelera la convergencia. También se optimiza el manejo de la resolución con el parámetro *output-stride*, el cual indica la resolución del mapa de segmentación antes del *upsampling*, que sigue permitiendo valores de 8 o 16, pero con una mayor flexibilidad gracias a la combinación de ASPP mejorado y *Multi-Grid*.

Finalmente, una de las diferencias más notables con DeepLab v2 es la eliminación del CRF denso en la fase de inferencia. Gracias a la mejora en la capacidad de segmentación de la red, ya no es necesario refinar los bordes mediante un modelo probabilístico externo, lo que simplifica la arquitectura y reduce el tiempo de inferencia. Como resultado, DeepLab v3 logra una segmentación más precisa y eficiente.

En cuanto al entrenamiento este es igual que el del modelo anterior.

El diagrama que se observa a continuación ilustra el diseño del modelo DeepLab v3:

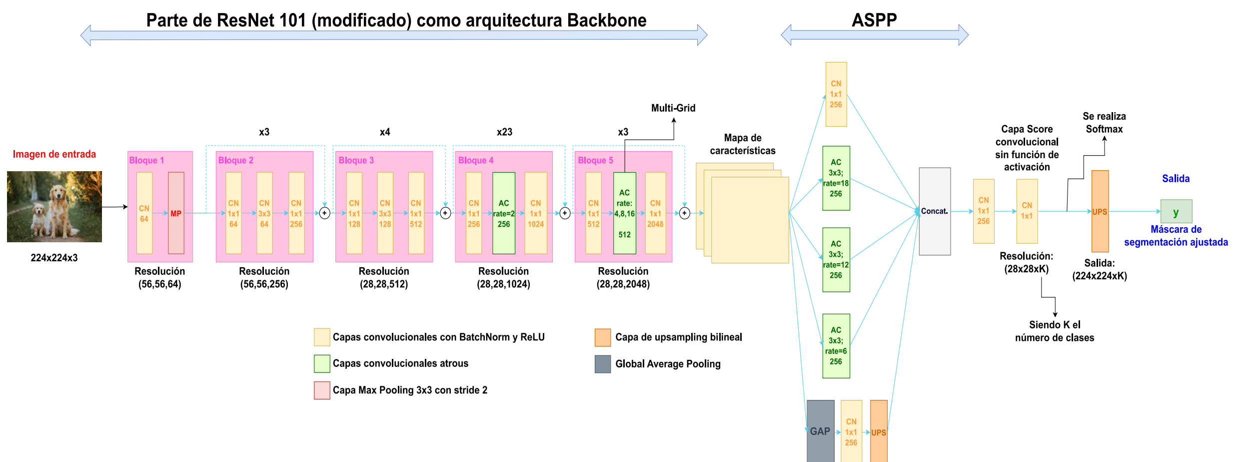



*Figura 8. Arquitectura de DeepLab v3*

### 3.1.2.4 Explicación de DeepLab v3+

Presentado en 2018 DeepLab v3+ [15] es la última versión oficial de la familia DeepLab. Combina lo mejor de DeepLab v3 (contexto multiescala) con un decodificador ligero para refinar detalles espaciales, logrando precisión en bordes y mayor eficiencia.

DeepLab v3+ introduce una arquitectura codificador-decodificador que combina lo mejor de su predecesor, DeepLab v3. A diferencia de DeepLab v3,

que se centraba únicamente en capturar contexto multiescala mediante el módulo ASPP mejorado y *Multi-Grid*, DeepLab v3+ integra un decodificador ligero que refina la segmentación al fusionar características de alta resolución de capas intermedias del codificador con características semánticas profundas tras el procesado de ASPP. Este enfoque permite preservar bordes precisos y detalles pequeños.

Una innovación clave de DeepLab v3+ es el uso de convoluciones separables *atrous* (*atrous separable convolutions*), que reducen drásticamente la complejidad computacional sin sacrificar rendimiento. Estas convoluciones descomponen la operación estándar en dos pasos: una convolución *depthwise* que es como una convolución normal, pero en vez de aplicar un filtro para todos los canales a la vez, utiliza un filtro distinto para cada canal (esta convolución *depthwise* en verdad es *depthwise atrous* porque en DeepLab v3+ el filtro a cada canal es *atrous*) y una convolución *pointwise* (combinación de canales con filtros 1x1), que no necesariamente reduce la dimensionalidad, es básicamente aplicar una convolución 1x1 a la salida *depthwise*. Además, DeepLab v3+ admite *backbones* flexibles, como Xception [51] el cual es modificado en el artículo original dado que se sustituyen las capas *max pooling* por las *atrous separable convolutions*.

El decodificador de DeepLab v3+ opera en tres etapas: primero, realiza un *upsampling* x4 del mapa de características del codificador; luego, concatena estas características con las de una capa intermedia del codificador, que conservan detalles espaciales de mayor resolución; finalmente, aplica convoluciones separables para fusionar la información y un *upsampling* final x4. Este proceso contrasta con DeepLab v3, que carecía de un decodificador y dependía únicamente de interpolación bilineal, limitando su capacidad para reconstruir bordes precisos.

En la próxima imagen se esquematiza la estructura general y los elementos clave del modelo DeepLab v3+:

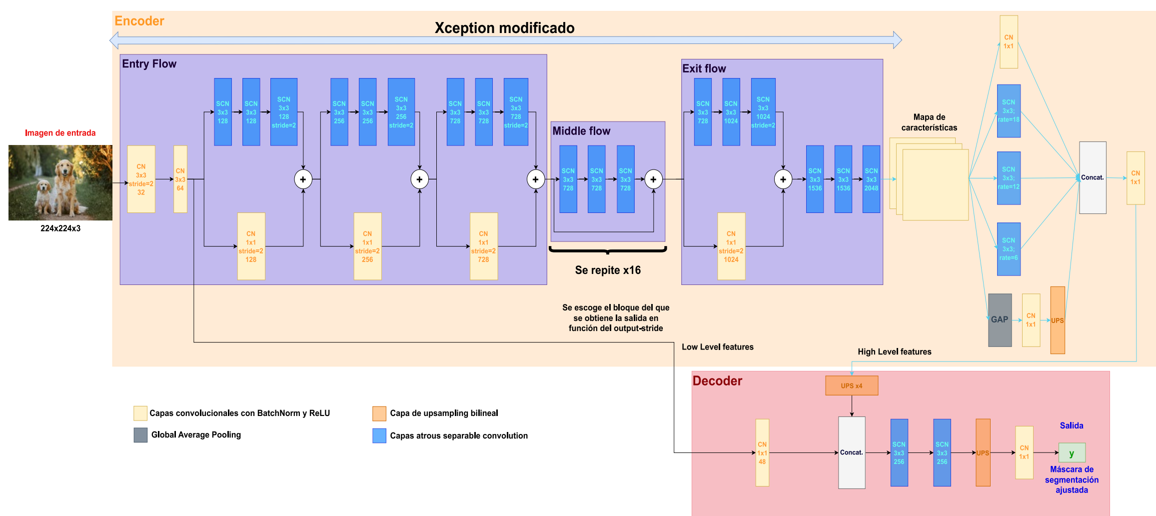



*Figura 9. Arquitectura de DeepLab v3+*

## 3.2 Modelos empleados para segmentación por instancia

Este apartado se centra en la segmentación por instancia, un proceso que no solo identifica las clases presentes en la imagen, sino que también diferencia entre múltiples instancias de un mismo objeto. Se examina en detalle la arquitectura de Mask R-CNN (una de las más relevantes en este campo), destacando sus mecanismos para la detección y separación de instancias individuales.

### 3.2.1 Mask R-CNN

La arquitectura que establece los fundamentos de la segmentación por instancia es Mask R-CNN [27], una red cuyo artículo fue publicado en 2017 por Kaiming et al. Este modelo se construye sobre la base de Faster R-CNN , una arquitectura previa que, a su vez, deriva de los modelos anteriores Fast R-CNN y R-CNN.

#### 3.2.1.1 R-CNN

Para comprender al completo el funcionamiento de Mask R-CNN, se debe analizar la evolución de las arquitecturas precursoras mencionadas. En este contexto, el punto de partida lo constituye R-CNN (*Region-based Convolutional Neural Network*), un modelo introducido por primera vez a finales de 2013, desarrollado por Girshick et al [52].

Este modelo aún queda bastante lejos de la segmentación por instancia, pero da los primeros pasos hacia la detección a nivel de píxeles en una imagen al detectar objetos en imágenes mediante cajas propuestas.

**PARTE 1:**

En términos generales, el modelo recibe una imagen y realiza dos tareas principales: clasifica los objetos presentes en categorías predefinidas (como "persona", "vehículo" o "animal") y los enmarca mediante *bounding boxes* (cajas delimitadoras). Para comprender cómo se lleva a cabo esta detección de manera eficiente, es necesario abordar un desafío clave: el elevado coste computacional que supone analizar todas las posibles regiones de una imagen.

Aquí radica la relevancia del término incluido en el nombre del modelo, *"Region-based"* (basado en regiones). En lugar de evaluar infinitas regiones, el enfoque se centra en buscar regiones de interés (ROIs o *Regions of Interest*, por sus siglas en inglés) para ser analizadas, que son segmentos específicos de la imagen con alta probabilidad de contener objetos. Estas ROIs son esenciales para optimizar el proceso, ya que permiten al modelo priorizar áreas relevantes y evitar el análisis de toda la imagen.

Para identificar estas regiones, el modelo emplea un algoritmo propuesto por Uijlings et al. en un artículo de 2012 [53], conocido como búsqueda selectiva. Este método consta de dos etapas fundamentales:

**1) Generación de una segmentación inicial** de la imagen mediante el uso de la técnica "Segmentación eficiente de imágenes basada en gráficos" propuesta por Felzenszwalb y Huttenlocher [54] consistente en la segmentación de una imagen en regiones más pequeñas. Esta técnica busca tratar los píxeles de una imagen como un grafo G=(V,E) donde V son los vértices (en este caso cada píxel) y E son las aristas formadas por ese píxel con sus píxeles vecinos, considerándose dos casos de píxeles vecinos: $N_4$ *neighborhood* en el cual se consideran vecinos los píxeles de las direcciones principales (arriba, abajo, izquierda y derecha) y $N_8$ *neighborhood* en el que se consideran también las direcciones diagonales. La base de la técnica es:

**1.1) Inicialización:** Al inicio se considera a cada vértice como un componente individual C perteneciente a un conjunto de componentes, S=[C1, C2,...,Cn].

**1.2) Unificación de componentes:** Una vez se tienen los componentes iniciales se deben ir unificando entre sí los que se puedan unificar, para formar componentes más grandes. Para ello se utilizan las siguientes expresiones:

$$Dif(C_x, C_y) = \min_{y \in C_x,\ V_y \in C_y,\ (V_y,\ V_x) \in E} W(V_x, V_y),$$

$Dif(C_x, C_y)$ indica la diferencia entre dos componentes (que pueden contener varios píxeles), la cual se calcula como la arista con menor peso (la que tiene menor diferencia de intensidades entre los píxeles que la conforman) que conecta un píxel de un componente con otro píxel del otro componente. En la expresión $W(V_x, V_y)$ indica que tan diferentes son dos píxeles, usando como diferencia la diferencia absoluta entre sus intensidades (donde el valor absoluto asegura una diferencia positiva).

Otra expresión empleada es:

$$Int(C_x) = \max_{e \in MST(C_x, E)} w(e),$$

donde MST es "*Minimum Spanning Tree*" o Árbol de Expansión Mínima que en el caso $MST(C_x, E)$ indica cuales son los Edges (E) del componente $C_x$ que forman el camino de menor valor que no produce bucles y permite llegar a todos los vértices del componente. Sabiendo esto $Int(C_x)$ es la arista del MST del componente $C_x$ con mayor peso (mayor diferencia de intensidades). Al principio del todo cuando los componentes solo contienen un vértice $Int(C_x)$ es igual a 0.

También se debe conocer la expresión de τ:

$$\tau(C_x) = \frac{K}{|C_x|},$$

siendo $|C_x|$ el cardinal del componente $C_x$ (número de vértices que contiene) y K un número cualquiera escogido (un mayor K proporciona componentes más grandes).

Esto último se aplica en la siguiente expresión:

$$MInt(C_x, C_y) = min\left(Int(C_x) + \tau(C_x), Int(C_y) + \tau(C_y)\right).$$

Una vez conocidas las expresiones anteriores ya se puede decidir si dos componentes se pueden unificar mediante la siguiente expresión:

$$D(C_x, C_y) = \begin{cases} True, & si\ Dif(C_x, C_y) > MInt(C_x, C_y) \\ False, & en\ caso\ contrario \end{cases},$$

siendo $D(C_x, C_y)$ lo que indica si dos componentes se pueden unificar (cuando $D(C_x, C_y) = False$ $C_x$ y $C_y$ se pueden unificar).

Un ejemplo del funcionamiento del algoritmo de segmentación eficiente de imágenes basada en gráficos se puede ver en el apéndice A.2.

En el caso de imágenes RGB el algoritmo es el mismo, en el artículo original se indica que para imágenes a color se debe ejecutar el algoritmo en cada canal, unificando únicamente los píxeles vecinos cuyos tres canales se encuentren en el mismo componente (intersección entre canales).

Esta primera parte del algoritmo de búsqueda selectiva proporciona una imagen segmentada como la que se ve en la siguiente figura:

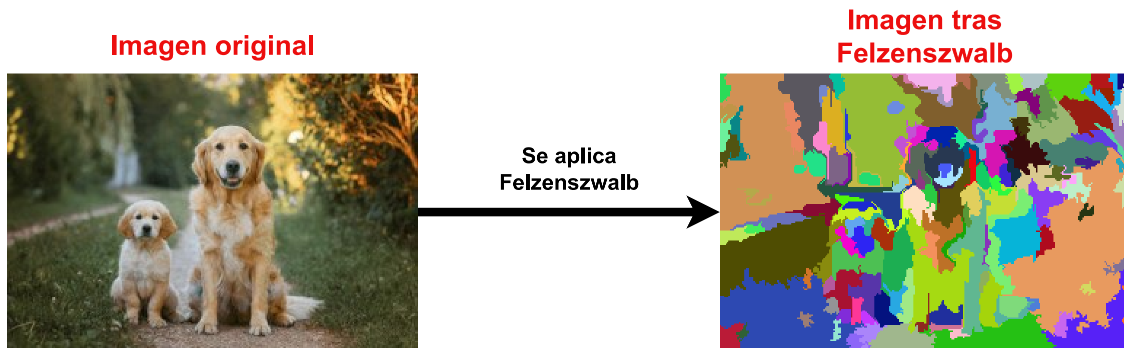



*Figura 15. Visualización de imagen tras aplicar Felzenszwalb*

**2)** La siguiente parte del algoritmo de búsqueda selectiva consiste en seguir unificando las regiones obtenidas anteriormente en regiones más grandes con el objetivo de reducir el número de regiones de interés totales. Para esta unificación se utiliza el algoritmo voraz en el siguiente orden:

1. Primero del conjunto de regiones obtenidas en el paso anterior se escogen de dos en dos las vecinas más similares.
2. Se unifican dichas regiones.
3. Se repiten los pasos anteriores múltiples veces.

Esto se puede representar en pseudocódigo de la siguiente manera:

a) Cálculo de la similitud entre regiones vecinas:

- *Efficient Graph-Based Image Segmentation* --> Regiones iniciales: $R=\{r_1,r_2,\ldots,r_n\}$

- Inicialización del conjunto de similitudes $S = \emptyset$

```
foreach(par de regiones vecinas: (rᵢ,rⱼ)) do{
    -  Cálculo de similitud entre ambas regiones s(rᵢ,rⱼ)
    -  S=S ∪ s(rᵢ,rⱼ) (se introducen las similitudes calculadas en S)
}
```

b) Unificación de las regiones más similares:

```
while(S≠∅){
    - Se toman las dos regiones más similares de S: s(rᵢ,rⱼ)=máx(S)

    - Se unifican en una nueva: rₜ=rᵢ ∪ rⱼ

    - Se elimina cualquier similitud del conjunto S que contenga
    alguna de las dos regiones unificadas:

        •  Respecto rᵢ: S=S\s(rᵢ,r*)
        •  Respecto rⱼ: S=S\s(r*,rⱼ)

    - Se calcula la similitud entre la nueva región rₜ y sus vecinos
    (Sₜ), y dichas similitudes se meten en el conjunto S:

        •  S=S ∪ Sₜ
        •  R=R ∪ Rₜ
}
```

Con respecto al cómo se calcula la similitud entre regiones, en el artículo original tienen en cuenta varios factores, como qué tan similares son en: textura, color, tamaño y relleno. A continuación, se muestra el proceso del cálculo de cada una:

- Para el cálculo de la similitud en la textura los autores del artículo mencionan que hacen uso de derivadas Gaussianas en 8 orientaciones [55-58] aplicadas sobre una imagen (como un filtro). Esto permite obtener lo que se denomina la respuesta del filtro. Para más información acerca de cómo realizar derivadas Gaussianas con orientación consultar el apéndice A.3.

Una vez obtenidas las respuestas del filtro para cada dimensión en las 8 orientaciones ya se puede obtener la similitud de textura. Para ello se realiza un histograma de textura que básicamente es un histograma con 10 *bins* para cada orientación de cada dimensión, lo que supone que si se agrupan todas las *bins* de todos los histogramas en un único *array*, se obtiene un histograma final de n=8x3x10=240 de dimensionalidad $T_i = \{t_i^1, \ldots, t_i^n\}$. Con este histograma de textura calculado ya se puede obtener la similitud de textura siguiendo la siguiente expresión:

$$s_{texture}(r_i, r_j) = \sum_{k=1}^{n} \text{mín}\,(t_i^k, t_j^k).$$

- Para el cálculo de la similitud de color entre regiones se realizan histogramas de las intensidades de los píxeles en sus tres canales con 25 *bins*. La similitud en este caso es:

$$s_{color}(r_i, r_j) = \sum_{k=1}^{n} \text{mín}\,(c_i^k, c_j^k).$$

- Para la similitud de tamaño se comparan los tamaños de las regiones adyacentes:

$$s_{tamaño}(r_i, r_j) = 1 - \frac{tamaño(r_i) + tamaño(r_j)}{tamaño(imagen\ total)}.$$

- Y para la similitud de relleno se mide cuan de bien se superpone una región a otra. Esta superposición es:

$$s_{relleno}(r_i, r_j) = 1 - \frac{tamaño(\beta\beta_{ij}) - tamaño(r_i) - tamaño(r_j)}{tamaño(imagen\ total)},$$

donde la parte del numerador de la fracción indica el área que no superpone ni a $r_i$ ni a $r_j$.

- Una vez calculadas las similitudes anteriores se puede calcular la similitud total entre dos regiones adyacentes como:

$$s_{final}(r_i, r_j) = s_{color}(r_i, r_j) + s_{tamaño}(r_i, r_j) + s_{textura}(r_i, r_j) + s_{relleno}(r_i, r_j).$$

Con estos pasos anteriores ya sí se obtienen las ROIs con las que se entrena al modelo R-CNN, que aproximadamente serán unas 2000.

**PARTE 2: REENTRENAMIENTO**

Con respecto a la otra parte del nombre del modelo, CNN, esta se debe a que el modelo tiene como base extraer características de las ROIs y para ello utiliza una red neuronal convolucional. Concretamente en la mayoría de las implementaciones esta CNN es el modelo AlexNet [59] que ya ha sido entrenada con un *dataset* de diversas imágenes como lo es ImageNet [60] y que ahora se busca reentrenar con las imágenes de las que luego se detectarán los objetos. Este reentrenamiento se denomina *fine tuning* y se hace sobre las ROIs. AlexNet consta de la siguiente arquitectura:

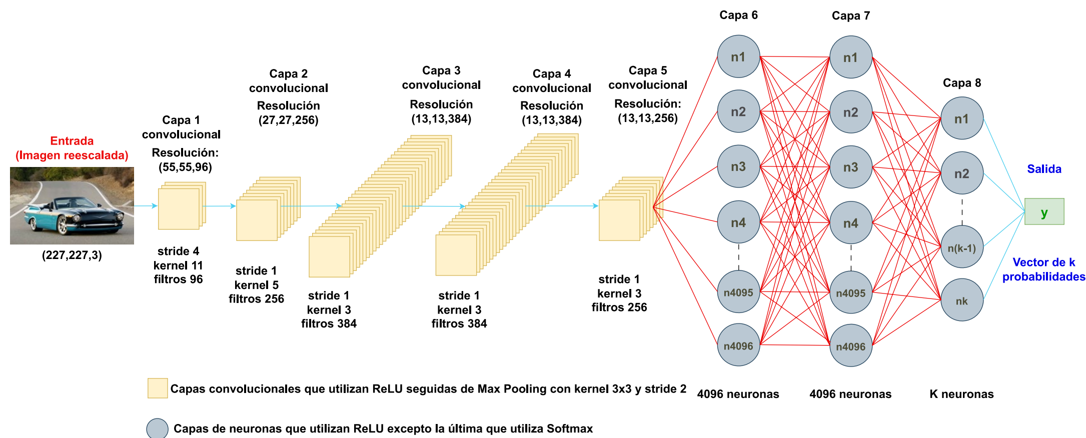



*Figura 16. Arquitectura de AlexNet*

Lo que se hace es reentrenar al modelo antes del entrenamiento de detección de objetos, a que únicamente indique a que clase pertenece cada ROI. Dicha predicción la realiza la última capa de la red. De las aproximadamente 2000 ROIs generadas se debe etiquetar a las regiones como pertenecientes a una clase K si su IoU (*Intersection over Union*, una métrica que mide la superposición entre dos áreas dividiendo el área común entre ambas por el área total que ocupan) con la caja de verdad de esa clase (las cajas del *dataset* de entrenamiento que contienen 4 coordenadas, Alto, Ancho, Centro en x y Centro en y, y que indican donde verdaderamente está el objeto) es ≥ 0.5, y como fondo las demás. Una vez hecho el *fine tuning* con las ROIs se elimina la última capa de clasificación (la capa 8) y se añaden nuevas capas a la arquitectura.

**PARTE 3: ENTRENAMIENTO**

Las nuevas capas que se encuentran en paralelo son dos:

**- Clasificador SVM (*Support Vector Machine*)**: El SVM es un tipo de algoritmo de aprendizaje supervisado que se usa para clasificar. Lo que hace SVM es separar datos en dos clases mediante un hiperplano que maximiza el margen entre ellas. En caso de que se tengan más de dos clases que separar se pueden utilizar dos estrategias:

- *One* vs *One*: En este enfoque se entrena un SVM para cada par de clases, cubriendo todas las combinaciones posibles. Por ejemplo, si tenemos 3 clases (A, B y C), se entrenan tres clasificadores:
  - Clasificador 1: A vs B
  - Clasificador 2: A vs C
  - Clasificador 3: B vs C

  Cuando llega un dato de la clase C, cada clasificador emite su predicción:

- Clasificador 1: Comprueba si el dato pasado C se parece un poco más a A o a B. Si se parece más a la B dirá la B.
- Clasificador 2: Reconoce directamente C y vota por C.
- Clasificador 3: También reconoce C y vota por C.

En este caso como la clase C es la que más veces se ha repetido se decide seleccionar la C como clase correcta. Lo malo de este método es que requiere entrenar a $\frac{N*(N-1)}{2}$ clasificadores siendo N el número de clases distintas a identificar.

- *One* vs *All*: En este enfoque solo se necesitan N clasificadores. Lo que se hace es entrenar a cada SVM para distinguir una clase como positiva y el resto como negativas:
  - Clasificador 1: A vs (B ∪ C)
  - Clasificador 2: B vs (A ∪ C)
  - Clasificador 3: C vs (A ∪ B)

  De igual manera que en el ejemplo del otro enfoque, suponiendo que se tienen datos de la clase C cada clasificador se comporta de la siguiente manera:
  - Clasificador 1: El primer clasificador asigna con baja probabilidad/puntuación los datos a la clase A, lo que significa "No A", por tanto, (B o C)
  - Clasificador 2: Pasa lo mismo, pero con la clase B, se asignan con baja probabilidad/puntuación los datos a la clase B ("No B"), por tanto, (A o C)
  - Clasificador 3: En este caso se asignan con alta probabilidad/puntuación los datos a la clase C.

  Como la que más puntuación tiene es la clase C, se escoge como correcta la clase C. Aunque este enfoque es más eficiente que el otro, puede ocurrir que un clasificador este sesgado hacia la clase mayoritaria, por ejemplo, si hay más imágenes de A que de B o C, puede que el modelo tienda a predecir los datos como pertenecientes a la clase A.

En R-CNN el SVM recibe como entrada el vector de características extraído de cada ROI por AlexNet (la salida de la última capa totalmente conectada, aplanada) y aprende a distinguir entre fondo y objetos de clase K según el grado de solapamiento con las *ground-truth boxes*. Concretamente:

- **Negativas (fondo):** ROIs con IoU < 0.3 respecto a cualquier caja de verdad.
- **Positivas (clase KKK):** ROIs con IoU = 1. El resto de ROIs (0.3 ≤ IoU < 1) se descartan y no participan en el entrenamiento.

En implementaciones prácticas suelen considerarse positivas también las ROIs con IoU ≥ 0,7, pero el artículo original solo toma las que alcanzan IoU = 1.

Finalmente, el SVM optimiza la función de pérdida *hinge* que compara la etiqueta real (fondo o una de las clases disponibles) con la predicción del clasificador, produciendo una puntuación de pertenencia a cada clase (incluido el fondo).

**- Regresor del cuadrado delimitador:** El regresor se emplea para afinar la localización de las ROIs propuestas por el algoritmo de búsqueda selectiva, ajustando sus coordenadas mediante un modelo de regresión lineal. Su objetivo es transformar cada ROI inicial en una caja delimitadora más precisa,

aprendiendo a predecir cuatro valores de desplazamiento (*targets*) ($t_x$, $t_y$, $t_w$, $t_h$) que permiten reconstruir la caja de verdad correspondiente.

Recalcula las coordenadas de cada ROI de la siguiente forma:

1. **Cálculo de coordenadas propuestas:**
   - Para cada ROI se extraen sus propiedades iniciales:

     $P_x$, $P_y$ = coordenadas del centro;

     $P_w$, $P_h$ = ancho y alto de la caja.
2. **Selección de ejemplos positivos:**
   - Se comparan las ROIs con las cajas de verdad (*ground-truth*) de la imagen.
   - Se calcula el IoU y se conservan únicamente aquellas ROIs con IoU ≥ 0,6 (umbral de entrenamiento). Estas se utilizan para entrenar al regresor, las demás se descartan.
3. **Cálculo de los valores target ($t_x$,$t_y$,$t_w$,$t_h$):**

   Cuando una ROI supera el umbral, se generan los desplazamientos que el regresor debe aprender para aproximar la caja de verdad ($G_x$, $G_y$, $G_w$, $G_h$):

   $$G_x = P_w * t_x + P_x$$
   $$G_y = P_h * t_y + P_y$$
   $$G_w = P_w * \exp(t_w)$$
   $$G_h = P_h * \exp(t_h)$$

Con respecto a lo que recibe el regresor, recibe lo mismo que SVM, un vector de características, y aprende mediante una capa de neuronas a producir las salidas, 4 salidas (los cuatro *targets*), por cada ROI. Para el aprendizaje del regresor se utiliza la función de pérdida *smooth* L1:

$$smooth\ L_1(x) = \begin{cases} 0.5 * x^2, si\ |x| < 1 \\ |x| - 0.5, si\ |x| \geq 1 \end{cases}$$

$$L_{reg} = \sum_{i \in \{x,y,w,h\}} smooth\ L_1(t_i^{pred} - t_i^{true})$$

**PARTE 4: NMS**

Para que la detección devuelva las ROIs más relevantes aún se deben eliminar todas las ROIs que son válidas, pero que se superponen sobre una misma región. Para ello los autores utilizan el método *Non-Maximum Suppresion* (NMS). NMS elimina las cajas de predicción más redudantes de la siguiente manera:

- Con lo devuelto por el SVM, un vector de probabilidades de pertenencia de cada ROI a cada una de las K clases, para cada ROI se selecciona la clase de mayor probabilidad y se agrupan todas las ROIs que comparten esa clase. Dentro de cada grupo, se ordenan las ROIs de mayor a menor confianza (probabilidad).

- Una vez ordenadas se toma la ROI con la confianza más alta y se compara su IoU con el resto de ROIs en el mismo grupo. Cualquier ROI cuyo IoU con la de mayor confianza exceda un umbral > 0.5 se elimina (la de menor confianza). Una vez comparada con todas, se pasa a la siguiente ROI en la lista (segunda de mayor confianza), esto se hace porque no se debe dejar únicamente las ROIs con más confianza para una clase ya que podría haber varios objetos de la misma clase en una imagen. El proceso se repite con las ROIs que aún permanecen. Y por último, se continúa hasta que no quedan pares de ROIs que comparar. A continuación se muestra un ejemplo:

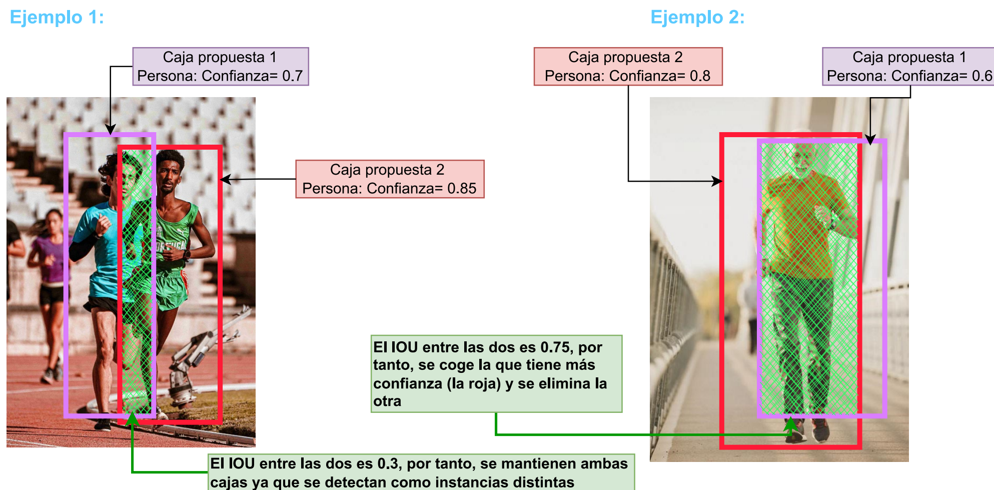



*Figura 17. Esquema del funcionamiento de NMS parte 1*

Por último, se debe tener en cuenta que si hay en una misma zona dos cajas de verdad superpuestas que aluden a clases distintas, si nuestra ROI se encuentra entre ambas, se le debe asignar como etiqueta correcta la que tenga mayor IOU, como se muestra en la siguiente figura:

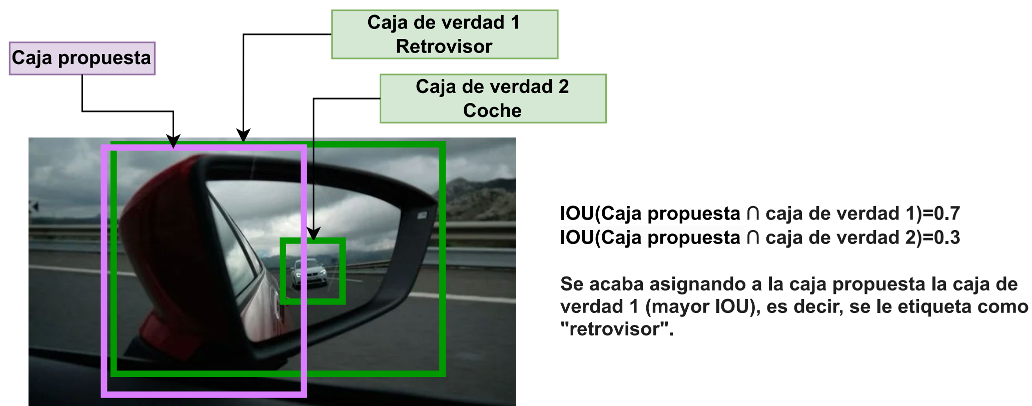



*Figura 18. Esquema del funcionamiento de NMS parte 2*

**PARTE 5: FUNCIÓN DE PÉRDIDA**

La función de pérdida global del modelo R-CNN es el sumatorio de la pérdida del clasificador y la pérdida del regresor:

$$L_{total} = L_{clasificador} + L_{regresor}$$

En la siguiente figura se recoge toda la arquitectura del modelo mencionado:

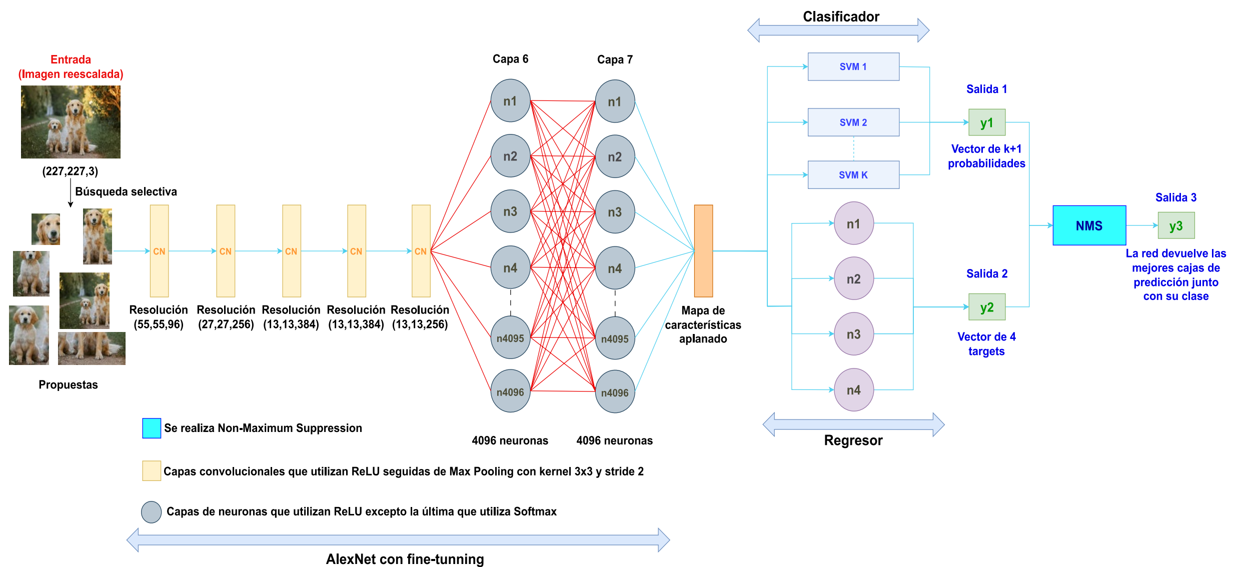



*Figura 19. Arquitectura de R-CNN*

### 3.2.1.2 Fast R-CNN

El elevado coste computacional que implica analizar tantas ROIs en R-CNN llevó a los autores del artículo de principios de 2015 Girshick et al. a proponer como solución el modelo Fast R-CNN [61].

En Fast R-CNN, en lugar de ejecutar inicialmente el algoritmo de búsqueda selectiva y posteriormente alimentar la CNN con las ROIs generadas, se introducen las imágenes originales directamente en la CNN, y se mapean las ROIs generadas por el algoritmo de búsqueda selectiva sobre los mapas de características que esta devuelve. De esta forma, en vez de analizar aproximadamente 2000 ROIs por imagen, la CNN solo debe aplicar las convoluciones sobre la imagen original.

Para determinar la posición que ocupa una determinada ROI en cada mapa de características, se dividen las coordenadas y las dimensiones originales por el factor de reducción que sufre la imagen a lo largo de la red, como se observa en la siguiente figura:

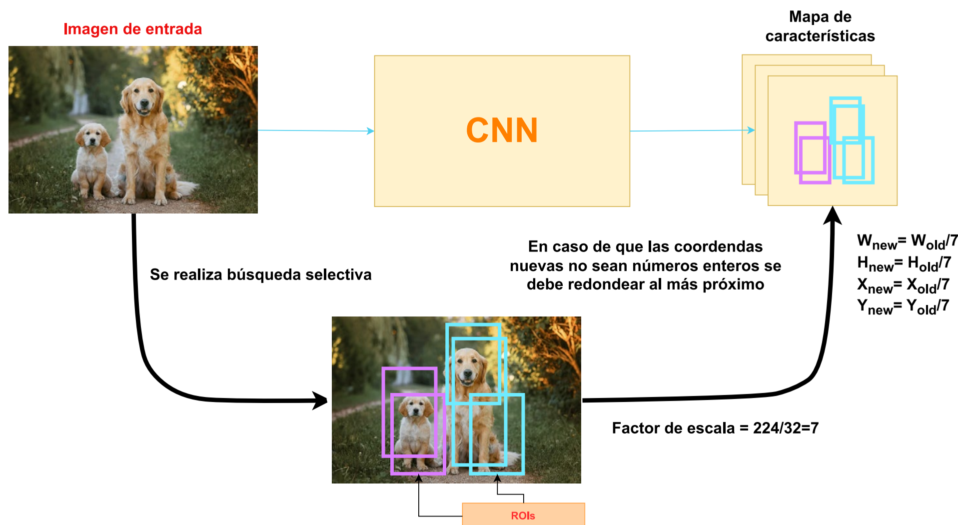



*Figura 20. Esquema del funcionamiento del mapeo de ROIs*

El inconveniente es que, ahora, las ROIs mapeadas sobre los mapas de características presentan dimensiones distintas, lo cual impide que puedan pasarse directamente a las siguientes capas. Por ello, los autores decidieron introducir el concepto de "ROI *Max Pooling*", que redimensiona las diversas ROIs mapeadas a una dimensión fija común. La capa de ROI *Max Pooling* opera subdividiendo la ROI en un número de partes equivalente a las dimensiones a las que se desea redimensionar la imagen, es decir, si se requiere que, tras aplicar esta capa, las regiones se redimensionen, por ejemplo, a 2x2, se debe dividir la ROI en 4 regiones (dos en horizontal y dos en vertical). Una vez dividida en estas regiones, se aplica el proceso de *Max Pooling* en cada una de ellas, es decir, se selecciona el valor máximo entre todos los píxeles de la región para asignarlo al píxel correspondiente en la nueva imagen redimensionada. Sin embargo, se debe seguir un criterio de división de la ROI para aquellos casos en los que las dimensiones de la ROI no sean múltiplos de las de la capa ROI *Max Pooling*, como se ilustra en el siguiente ejemplo:

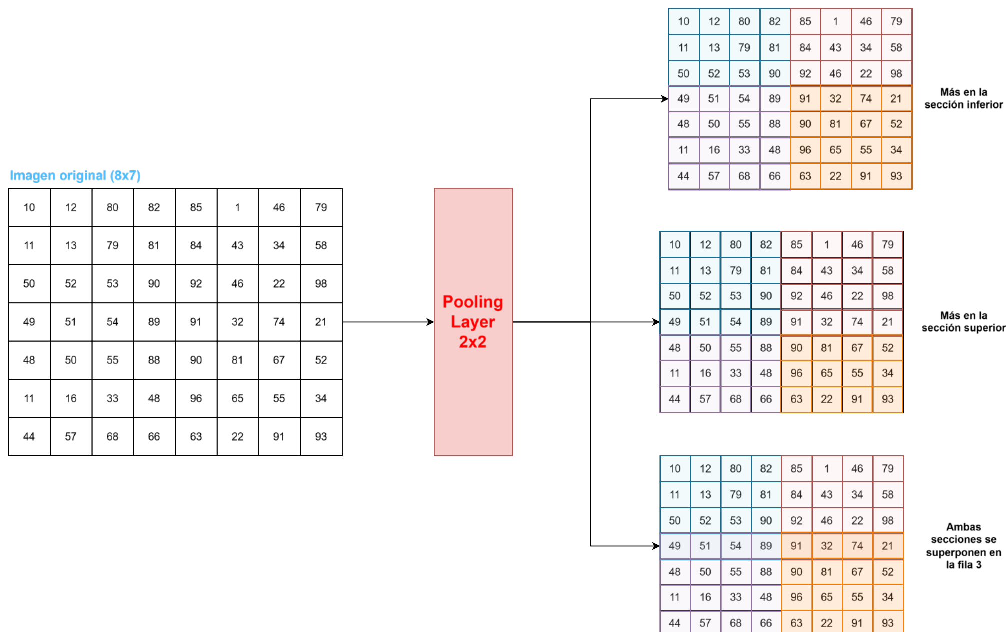



*Figura 21. Esquema de posibles casos al hacer ROI Max Pooling*

Como se observa en el ejemplo anterior la división vertical no supone ningún problema (2 es múltiplo de 8), sin embargo, la horizontal sí (2 no es múltiplo de 7) lo que implica que se puede dividir de diferentes formas. Un criterio que afronta este problema consiste en dividir la ROI en regiones de la siguiente manera:

- $h_{start} = \left[ j * \frac{roi_h}{pool_h} \right]$
- $h_{end} = \left[ (j+1) * \frac{roi_h}{pool_h} \right]$
- $w_{start} = \left[ i * \frac{roi_w}{pool_w} \right]$
- $w_{end} = \left[ (i+1) * \frac{roi_w}{pool_w} \right]$

Donde h y w representan que sección de la ROI se está cubriendo e i y j las coordenadas de la matriz nueva (tras hacer *ROI Max Pooling*) que contendrá los valores *max pooling* de cada región de la ROI, es decir, la coordenada (0,0) de la matriz contendrá el valor *max pooling* de la región correspondiente de la ROI.

Para entrenar esta red, al igual que en R-CNN, es necesario clasificar las ROIs generadas en positivas (las que pertenecen a una clase) y negativas (las que se consideran fondo). Para ello, los autores estiman la cantidad de ROIs a conservar (de las aproximadamente 2000 generadas) basándose en el número de imágenes originales que se utilizan simultáneamente durante el entrenamiento y en el tamaño del mini-batch en el que se incorporan dichas ROIs. Los autores recomiendan utilizar dos imágenes simultáneas y un mini-batch de 128, lo que implica conservar 128/2 = 64 ROIs de las aproximadamente 2000 disponibles. Como criterio de descarte, se seleccionan aleatoriamente ROIs hasta cumplir lo siguiente:

- Se asigna una etiqueta de clase (no fondo) a aquellas ROIs que tengan un IoU ≥ 0.5 con las cajas de verdad, garantizando que el 25% de las 64 ROIs seleccionadas sean positivas.
- Se asigna la etiqueta fondo a las ROIs que tengan un IoU < 0.5 con las cajas de verdad, de modo que el 75% de las ROIs sean negativas.

Tras aplicar la *Pooling layer* a las 64 ROIs, las matrices resultantes se aplanan para ser introducidas en las siguientes capas (dos capas dense en serie). Una vez se calcula el output de la última capa Dense, igual que en R-CNN la red se divide en dos partes:

**- Clasificadora:** En esta arquitectura, el clasificador evoluciona de utilizar un SVM (como en R-CNN) a emplear una capa lineal con una función de activación Softmax que predice la probabilidad de que una ROI pertenezca a cada una de las clases. Esto permite entrenar la red de forma *end-to-end* mediante una función de pérdida global que se propaga a través de toda la red. Para el clasificador se utiliza la función de pérdida *Cross-Entropy*, la cual compara el vector de predicciones de la salida Softmax para cada ROI con la etiqueta real (positiva o negativa).

**- Regresora:** El regresor funciona de igual manera que en R-CNN, entrenándose únicamente con las ROIs positivas.

La función de pérdida global utilizada es:

$$L_{total} = L_{clasificador} + \lambda[u \geq 1] * L_{regresor},$$

donde λ es un parámetro ajustable que determina el peso asignado al regresor. La variable u actúa de modo que, si es menor que 1, λ se anula (lo que implica que no se tenga en cuenta la pérdida del regresor) y, si es mayor o igual a 1, se asigna a la pérdida del regresor una importancia proporcional a λ. Es importante destacar que el regresor y el clasificador no comparten parámetros, por lo que, al calcular las derivadas parciales, el error del clasificador no influirá en el del regresor; sin embargo, ambos sí afectan a las capas compartidas previas, lo que indirectamente puede afectarles.

Durante el entrenamiento, se incorpora además el concepto de "*Scale Invariance*", cuyo objetivo es enseñar a la red especialmente a la parte convolucional a comprender cómo varían los objetos en una imagen cuando se modifican las escalas. Esto se consigue alimentando a la red con la misma imagen presentada en diferentes escalas. Un ejemplo de este proceso se muestra a continuación:

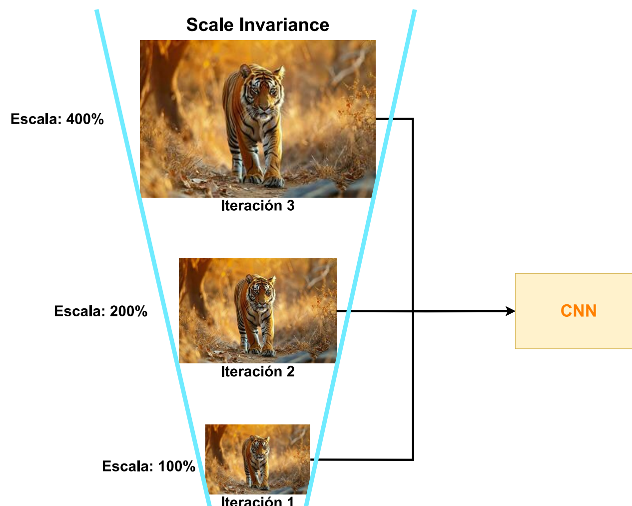



*Figura 22. Esquema del funcionamiento de Scale Invariance*

Por último, cabe mencionar que los autores de Fast R-CNN observaron que las capas densas anteriores al clasificador y al regresor consumían casi la mitad de la potencia computacional de la red, ya que estas procesan todas las ROIs. Para optimizar este aspecto, emplearon un método denominado "*Truncated SVD*", que, de manera simplificada, elimina los parámetros más irrelevantes en estas capas, reduciendo de forma drástica el tiempo de ejecución.

A continuación se presenta la arquitectura del modelo Fast R-CNN:

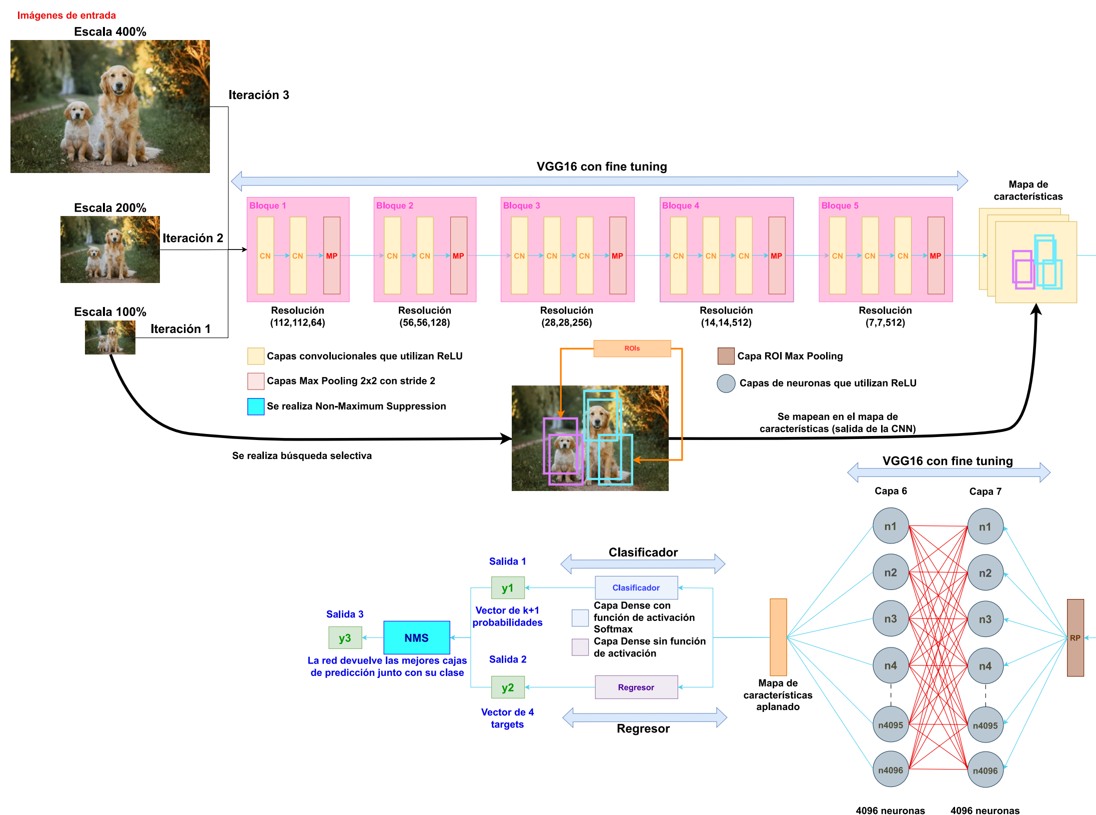



*Figura 23. Arquitectura de Fast R-CNN*

#### 3.2.1.3 Faster R-CNN

Más adelante, otro grupo de investigadores descubrió que era posible reducir aún más el tiempo requerido para la detección de objetos con Fast R-CNN. Este avance se presentó en el artículo publicado a mediados de 2015 escrito por Ren et al., con su arquitectura Faster R-CNN [62].

En dicho artículo se propone un cambio importante: eliminar el algoritmo de búsqueda selectiva que se utiliza para encontrar las ROIs. En su lugar, se plantea el uso de una red neuronal para predicir dichas ROIs, denominada RPN (*Region Proposal Network*).

La RPN se puede considerar como una red neuronal adicional que se sitúa entre la salida de la CNN y la ROI *Max Pooling*. En primer lugar, la RPN recibe el mapa de características producido por la CNN y le aplica una capa convolucional, cuyo resultado se bifurca en dos ramas:

**- Rama clasificadora:** Esta rama produce un vector que, para cada posición del mapa de características, asigna una probabilidad binaria (fondo 0 u objeto 1) a cada uno de los *anchors* (se explica a continuación). La salida tiene dimensiones (batch_size x H x W x Nº de ROIs, 1).

**- Rama regresora:** Paralelamente, esta rama ajusta las posiciones de las ROIs (propuestas clasificadas como "objeto") calculando 4 valores *target* que afinan la ubicación y forma de cada *anchor*. Su salida tiene dimensiones (batch_size x H x W x Nº de ROIs, 4).

Para generar las ROIs, en lugar de utilizar algoritmos externos (como *Selective Search*), se emplean los llamados "*anchors*" mencionados. Por cada píxel del mapa de características se generan 9 *anchors* obtenidos al combinar 3 escalas (por ejemplo, 128, 256 y 512) con 3 ratios de aspecto (2:1, 1:1 y 1:2). Así, para una escala de 128 se pueden obtener, por ejemplo, *anchors* de aproximadamente 181.02 píxeles de ancho y 90.51 píxeles de alto para el ratio 2:1; de 128x128 para el 1:1; y 90.51x181.02 para el 1:2. Estos cálculos aseguran que el área total de cada *anchor* sea constante, ya que 128x128 = 16384, al igual que 181.02x90.51 ≈ 16384.

Los valores del tamaño de los *anchors*, dado una escala y un ratio concreto, se estiman de la siguiente manera para mantener constante el número total de píxeles en cada *anchor*:

$$\begin{cases} 1)\ \dfrac{h}{w} = ratio\ (h\ y\ w\ son\ proporcionales) \\ 2)\ h * w = escala^2 \end{cases} \rightarrow \begin{cases} 1)\ h = ratio * w \\ 2)\ (ratio * w) * w = escala^2 \end{cases}$$

$$\rightarrow \begin{cases} 2)\ w^2 = \dfrac{escala^2}{ratio} \end{cases} \rightarrow \boldsymbol{w} = \sqrt{\dfrac{escala^2}{ratio}} = \dfrac{\boldsymbol{escala}}{\sqrt{\boldsymbol{ratio}}},$$

$$y, por\ tanto \rightarrow h * \dfrac{escala}{\sqrt{ratio}} = escala^2 \rightarrow \boldsymbol{h = escala} * \sqrt{\boldsymbol{ratio}}$$

Estos *anchors*, como se puede intuir por el ejemplo, se mapean sobre la imagen original y no sobre el mapa de características, lo que explica por qué sus dimensiones son tan grandes. Una vez definidos los distintos tipos de *anchors* a generar, es necesario calcular las posiciones que ocuparán en la imagen original, considerando que cada uno se centra en un píxel del mapa de características. Esto se realiza de la siguiente manera: si se desea que el *anchor* esté centrado en cada píxel, se calcula cuánto se extiende hacia la izquierda, hacia la derecha, hacia arriba y hacia abajo; en este caso se utiliza la notación

$(-\frac{w}{2}, -\frac{y}{2}, \frac{w}{2}, \frac{h}{2})$, que equivale a definir las coordenadas iniciales ($x_{inicial}$, $y_{inicial}$) y finales ($x_{final}$, $y_{final}$). De este modo, para un *anchor* con escala 128x128 y ratio 2:1, se ocupa la región de la imagen original (-90.51, -45.255, 90.51, 45.255) para el píxel 0 del mapa de características. Las coordenadas negativas también se proyectan sobre la imagen original, aunque posteriormente se recorta la parte que excede sus límites.

El cálculo anterior se efectúa para determinar la posición de los *anchors* correspondientes al primer píxel del mapa de características. Para ubicar los *anchors* de los píxeles restantes, es fundamental establecer la equivalencia entre un píxel del mapa y la cantidad de píxeles en la imagen original. Por ejemplo, si la imagen original mide 224x224 píxeles y el mapa es de 32x32, entonces cada píxel del mapa equivale a aproximadamente 7x7 píxeles en la imagen original (224/32 = 7). Así, el conjunto de 9 *anchors* se desplaza 7 píxeles en vertical (coordenadas y) o en horizontal (coordenadas x) cada vez que se avanza al siguiente píxel del mapa, garantizando la correcta proyección de los *anchors* sobre la imagen original. Con este procedimiento se obtienen todas las ROIs.

Por último se eliminan las ROIs/*anchors* más irrelevantes utilizando NMS y se eliminan todas las coordenadas negativas, recortando hasta estar dentro de los límites de la imagen.

La siguiente figura ilustra el esquema de la arquitectura de la RPN:

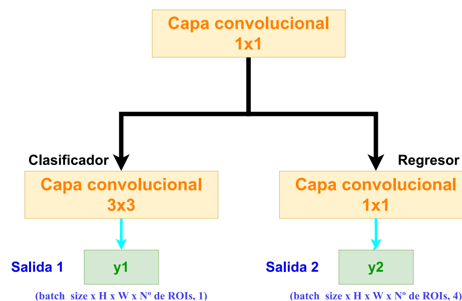



*Figura 24. Esquema de la arquitectura de RPN*

Una vez generados los *anchors*, se procede al entrenamiento de la RPN. Conociendo las cajas de verdad, se asigna a cada *anchor* una etiqueta de "objeto", "fondo" o "ignorar" utilizando el criterio siguiente:

- Si el IoU entre el *anchor* y la caja de verdad es ≥ 0.7 se etiqueta al *anchor* como objeto.
- Si 0.3 ≤ IoU < 0.7 se etiqueta al *anchor* como "ignorar".
- Si su IoU < 0.3 con la caja de verdad se etiqueta al *anchor* como "fondo".
- Además, si alguna caja de verdad no tiene asignado ningún *anchor*, se asigna el *anchor* con mayor IOU, aún si no cumple los criterios anteriores.

Para la rama clasificadora se utiliza la *Binary Cross-Entropy* (solo clasifica como fondo u objeto asignando una probabilidad a cada clase) entre los valores predicho por el clasificador y las verdaderas etiquetas, omitiendo los casos "ignorar".

En paralelo se calculan los valores *target* del regresor (los que debe predecir, estos se encuentran en el *dataset*) para cada *anchor*. Y posteriormente se obtiene la pérdida del regresor utilizando como función *smooth* L1, que calcula la diferencia entre lo que ha predicho el regresor y los verdaderos *target*, tanto para ejemplos "objeto" como objetos "fondo" (no se calcula la pérdida con objetos "ignorar").

La función total de pérdida de la RPN es la suma de la pérdida de su clasificador y la pérdida de su regresor:

$$L_{RPN} = L_{clasificador} + L_{regresor}.$$

El resto del modelo Faster R-CNN es igual en arquitectura que Fast R-CNN:

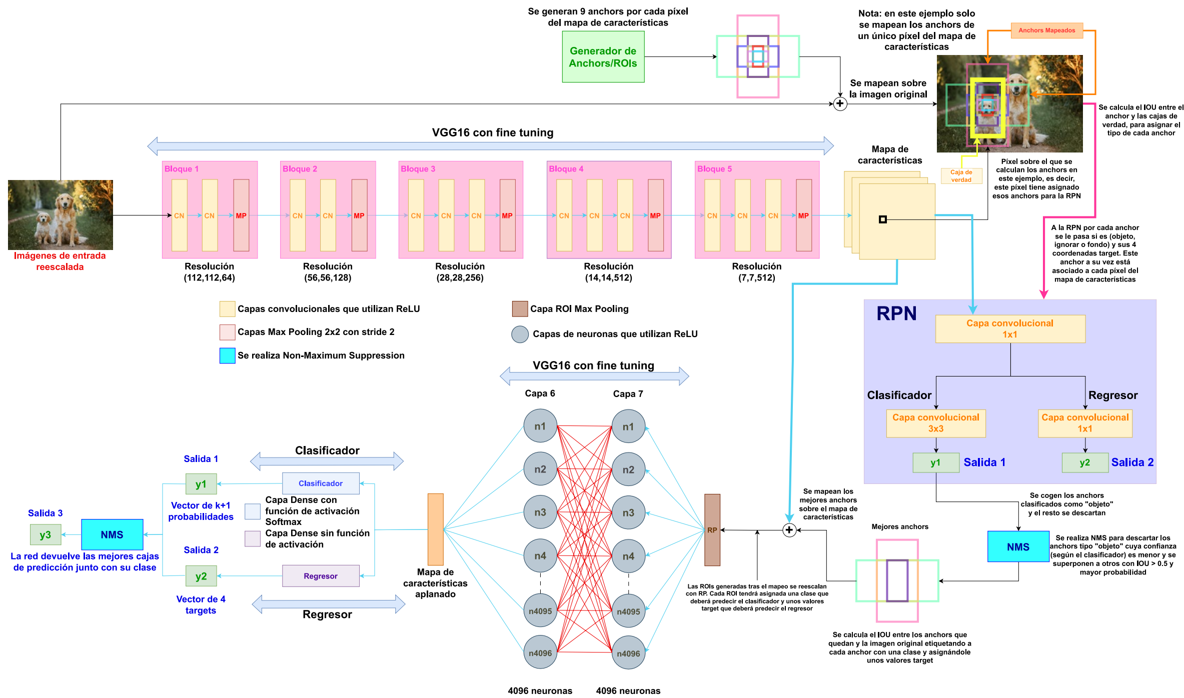



*Figura 25. Arquitectura de Faster R-CNN*

Como se observa en la figura, después de obtener las ROIs a partir de la RPN y aplicar NMS para retener las mejores propuestas, estas alimentan al resto de la red, que sigue la arquitectura de Fast R-CNN. Sin embargo, antes de continuar, es imprescindible mapear las ROIs sobre los mapas de características generados por la CNN. Es este mapeo el que permite emplear las ROIs ajustadas en las capas posteriores, de forma similar a cómo Fast R-CNN utilizaba las ROIs derivadas del algoritmo de búsqueda selectiva.

La función de pérdida total de la red es igual a la suma de todas sus pérdidas:

$$L_{total} = L_{RPN} + L_{clasificador} + L_{regresor}.$$

### 3.2.1.4 Explicación de Mask R-CNN

Mask R-CNN ya no es solo un modelo de detección de objetos como los anteriores, este modelo, además, permite realizar segmentación por instancia sobre las imágenes que recibe. Su arquitectura es similar a la de Faster R-CNN, pero incorpora varios cambios.

Primero en Mask R-CNN la CNN que hace de *backbone* no es una CNN como las utilizadas previamente. En este caso se utiliza la red ResNet, capaz de extraer características de imágenes a distintos niveles con mucho detalle. ResNet como ya se mostró es una arquitectura que genera mapas de características en diferentes resoluciones (C1, C2, C3, C4 y C5). Entre las distintas ResNet una de las más utilizadas es ResNet-50 la cual se presenta a continuación:

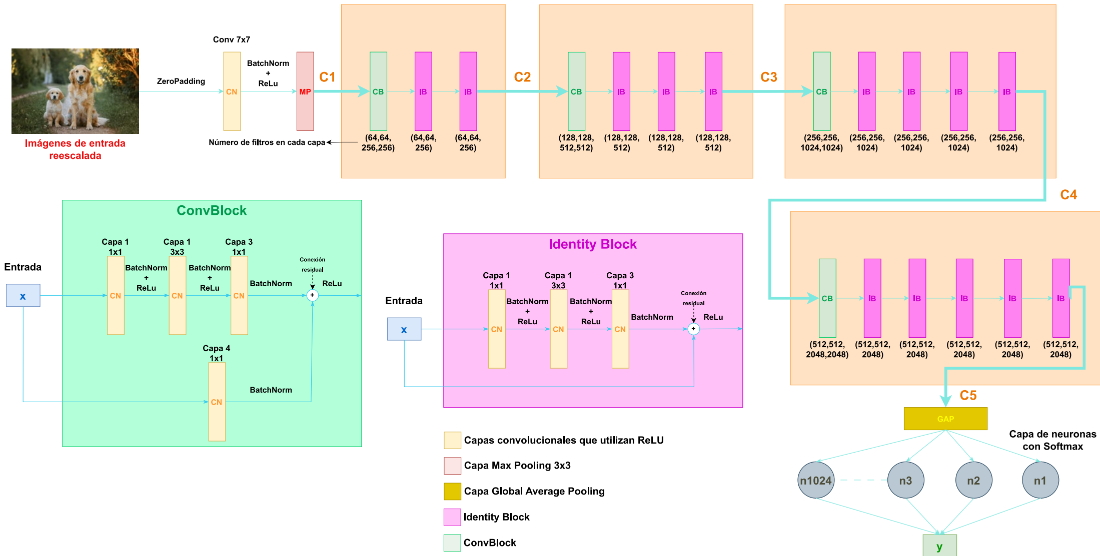



*Figura 26. Arquitectura ResNet-50*

Aunque también se utiliza la versión ResNet-101 en muchos casos.

Las distintas "C" mostradas en la figura se utilizan en Mask R-CNN para alimentar una FPN (*Feature Pyramid Network*) [63], que consiste en una arquitectura utilizada por el modelo, diseñada para mejorar la representación de características a diferentes escalas, de forma que se obtiene más información de la imagen. La FPN se muestra a continuación:

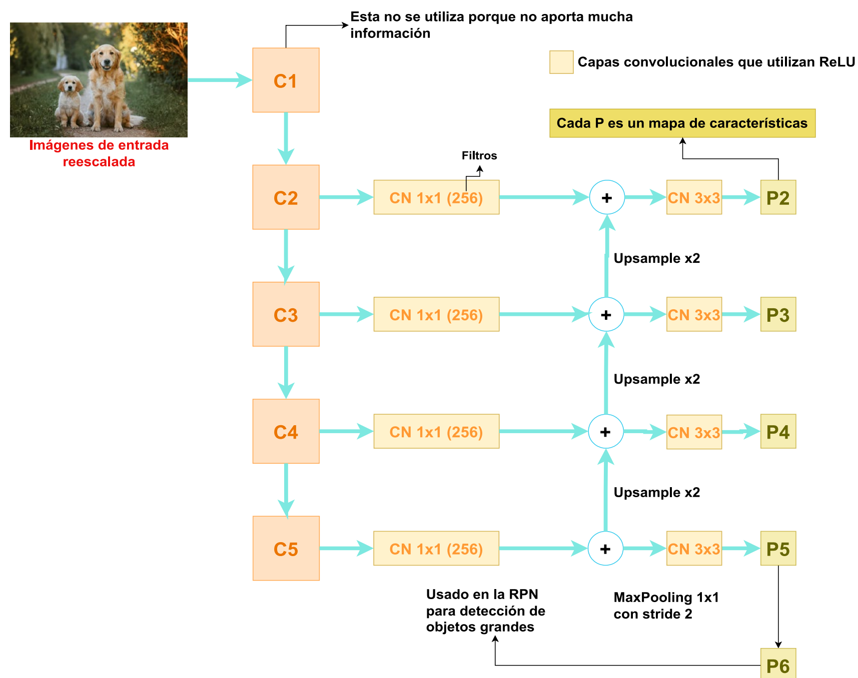



*Figura 27. Arquitectura de la FPN*

Una vez se tiene la FPN construida se utilizan RPNs por cada nivel, es decir, los niveles (P2, P3, P3, P4, P5 y P6) se utilizan para entrenar a seis RPNs distintas (P1 no se coge porque dicho nivel posee poca información útil). Cada RPN calcula con los mapas de características pasados por las capas P las etiquetas (clasificador) y los valores *target* (regresor). Posteriormente se hace NMS a las salidas de las RPNs para obtener las mejores ROIs. Y por último se realiza IoU entre las ROIs de cada RPN y las cajas de verdad, marcando (igual que en los anteriores modelos) como positivas y etiquetadas a una clase las ROIs con IoU $\geq 0.5$ y negativas, etiquetadas como fondo las IoU $< 0.5$. A continuación se muestra un esquema de lo mencionado:

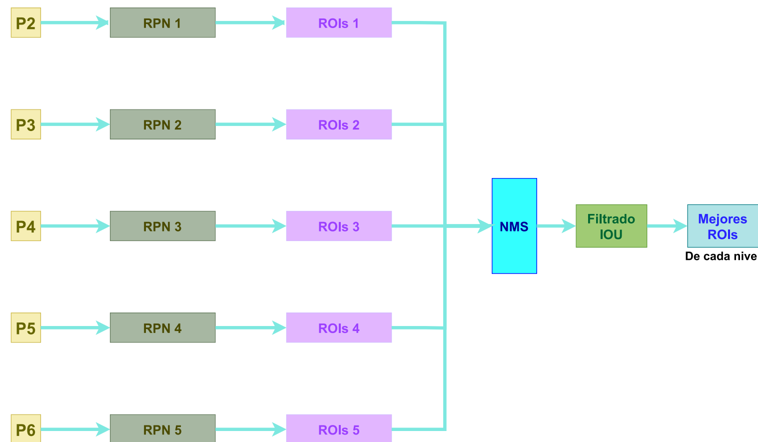



*Figura 28. Esquema de la conexión entre FPN y RPN*

Tras mapear las ROIs seleccionadas con los mapas de características provenientes de las salidas de P y seleccionar las mejores de cada nivel, la siguiente capa en la arquitectura sería la ROI *Max Pooling*. Sin embargo, los autores de Mask R-CNN notaron que este método eliminaba información importante debido a la cuantización inherente al *max pooling*. Para resolver este problema, introdujeron la capa "ROI *Align*", la cual garantiza que todas las ROIs tengan dimensiones uniformes. A diferencia del *max pooling*, ROI *Align* utiliza interpolación bilineal para muestrear los valores exactos de las características en puntos específicos dentro de cada celda de la matriz resultante, evitando así la pérdida de información causada por la cuantización. Un ejemplo del funcionamiento del ROI Align se puede ver en el apéndice A.4.

Esta técnica funciona tanto si la matriz ROI es cuadrada como si no y consigue como se ha comentado perder menos información que con ROI *Max Pooling*.

Cabe destacar que, en el proceso de ROI *Align*, no se utiliza directamente el nivel P6; en su lugar, las ROIs generadas en P6 se reasignan al nivel P5 antes de proceder con la alineación. Esto garantiza que a cada caja se le asigne un nivel específico para mapearla con su correspondiente mapa de características y, posteriormente, se realice ROI *Align*, generando imágenes cuadradas de dimensiones uniformes (por ejemplo, 7x7).

A partir de aquí, el flujo continúa de forma similar a Faster R-CNN, donde los resultados se pasan a las siguientes capas para clasificar las ROIs y predecir sus correspondientes *targets*. En el caso de Mask R-CNN, además se incorpora una cabeza adicional, denominada "*Mask Head*", encargada de la predicción de la máscara para realizar la segmentación por instancia.

El entrenamiento de la red sigue el mismo esquema que en Faster R-CNN, con la incorporación adicional del entrenamiento de la *Mask Head.* Esta cabeza devuelve N máscaras (una por cada clase) para cada ROI, de las cuales únicamente se utilizan aquellas que coinciden con la clase real asignada a la ROI. La pérdida para la máscara se calcula mediante *Binary Cross-Entropy*, comparando la máscara real con la predicha para cada ROI. De este modo, la función de pérdida total de la red se define como la suma de las pérdidas de cada componente del sistema:

$$L_{total} = L_{RPN} + L_{clasificador} + L_{regresor} + L_{mask\ head}$$

Para comprender el proceso completo, se resume a continuación el flujo de aprendizaje y la utilización de la salida de la red:

**- Fase de aprendizaje:**

**1) Obtención de los mapas utilizados en la RPN:** Al inicio, la red recibe una imagen del dataset que se envía al *backbone* ResNet. Este extrae mapas de características en cinco niveles (C1, C2, C3, C4 y C5). A continuación, estos mapas alimentan la FPN, que produce otras cinco salidas: P2 a P5, obtenidas a partir de C2–C5, y P6, generado aplicando *Max Pooling* sobre P5.

**2) Generación y entrenamiento de la RPN:** La RPN actúa simultáneamente sobre los mapas P2–P6, colocando N *anchors* en cada píxel (por ejemplo, tres escalas distintas por nivel). Su rama de regresión predice cuatro valores ($t_x$,$t_y$,$t_w$,$t_h$) por *anchor*, produciendo un tensor de forma (N×4,H,W,batch). La rama de clasificación etiqueta cada *anchor,* en función de cuál es su IoU con las cajas de verdad, como “objeto” ($IoU \geq 0{,}7$), “fondo” ($IoU < 0{,}3$) o “ignorar” ($0{,}3 \leq IoU < 0{,}7$). En cada iteración se emparejan los *anchors* con las cajas *ground-truth* para calcular dichos IoU, generando los targets reales para el regresor (solo para los positivos) y las etiquetas reales para el clasificador, con las que se calcula la pérdida de la RPN.

**3) Procesamiento de las ROIs y cabezas finales:** Tras aplicar NMS, ROI Align y filtrar por IoU (positivas: $IoU \geq 0{,}5$; negativas: $IoU < 0{,}5$), se seleccionan N propuestas positivas y M negativas. El regresor final predice nuevamente ($t_x$,$t_y$,$t_w$,$t_h$) para cada ROI, comparando únicamente las positivas con sus targets reales para calcular la pérdida de refinamiento de caja. El clasificador final produce, para cada ROI, un vector de tamaño K+1 (K clases más fondo), cuyas probabilidades se contrastan con la etiqueta real para evaluar la pérdida de clasificación. A su vez, la Mask Head genera una máscara H×W para cada clase y ROI; durante el entrenamiento solo se utilizan las máscaras de las regiones positivas y de la clase asignada, midiendo su error mediante *Binary Cross-Entropy* frente a la máscara *ground-truth.*

**4) NMS:** Finalmente, una última aplicación de NMS sobre las cajas, las probabilidades de clase y las máscaras predichas elimina solapamientos excesivos y conserva únicamente las detecciones más confiables.

**- Salida de la red tras el entrenamiento:**

- **Regresor final:** Para cada ROI positiva, el regresor predice los cuatro valores ($t_x$,$t_y$,$t_w$,$t_h$) que permiten refinar la posición y el tamaño de la caja. A partir de estos valores y de las coordenadas originales del anchor de mayor confianza, se calcula por transformaciones la caja predicha en el espacio de la imagen original.

- **Clasificador final:** Proporciona, junto con una medida de confianza (la utilizada en el regresor), la clase a la que pertenece cada ROI. Tras aplicar NMS, se retienen solo un número limitado de ROIs, cada una con su clase asignada y puntuación.

- **Máscara:** La Mask Head produce, para cada ROI y para cada clase, una máscara de tamaño reducido (por ejemplo, 28×28). Solo se mantiene la máscara correspondiente a la clase predicha por el clasificador. Esta máscara se interpola bilinealmente hasta encajar en la caja delimitadora refinada por el regresor, obteniendo la máscara de instancia final sobre la imagen original.

A continuación, se muestra de manera esquemática la arquitectura de Mask R-CNN:

*Figura 35. Arquitectura de Mask R-CNN*

# 4. Solución propuesta

El principal objetivo de este trabajo de fin de máster consiste en la implementación y comparativa de distintos modelos de segmentación semántica y por instancia, analizando sus características: U-Net, DeepLabv3+ y Mask R-CNN. Dicha comparativa se hace entrenando a los modelos hasta sus puntos de convergencia, por ello el número de *steps* de entrenamiento varía según el modelo ya que lo que se busca es ver cómo es su rendimiento máximo para un *dataset* concreto.

Para esta comparativa se ha hecho uso del repositorio de datos público de imágenes urbanas "*Cityscapes*" [64] que posee imágenes de 2048x1024 de resolución. Dicho repositorio cuenta con 2975 imágenes para el entrenamiento, 500 imágenes de validación y 1525 imágenes de test. Se debe aclarar que existe una distinción entre las clases semánticas que considera el *dataset* y las clases de instancia. De las semánticas se incluyen las siguientes:

**Clases semánticas (19 en total):**

Estas clases cubren tanto objetos contables (cosas) como regiones amorfas (materiales o estructuras) que no se segmentan como instancias individuales. Entre ellas se encuentran:

road – carretera; sidewalk – acera; building – edificio; wall – muro; fence – cerca; pole – poste; traffic light – semáforo; traffic sign – señal de tráfico; vegetation – vegetación; terrain – terreno; sky – cielo; person – persona; rider – conductor (de bicicleta o moto); car – coche; truck – camión; bus – autobús; train – tren; motorcycle – motocicleta; bicycle – bicicleta.

Sin embargo, no todas estas clases se utilizan en la tarea de segmentación por instancia. En ese caso, solo se consideran aquellas clases que representan objetos individuales, que se pueden contar y segmentar por separado.

**Clases de instancia (8 en total):**

Para la tarea de segmentación por instancia, Cityscapes anota las siguientes clases:

person – persona; rider – conductor; car – coche; truck – camión; bus – autobús; train – tren; motorcycle – motocicleta; bicycle – bicicleta.

Un ejemplo de imagen sacada del *dataset* de entrenamiento es el siguiente:

*Figura 36. Ejemplo de imagen del dataset de entrenamiento*

Para llevar a cabo el entrenamiento del modelo con menos coste computacional (U-Net) se utilizó una GTX 1660 TI, mientras que los otros modelos fueron entrenados en el entorno de Google Colab con las gráficas A100 de Nvidia ya que tienen mayor memoria VRAM y potencia de cómputo.

## 4.1 Arquitectura U-Net

Toda la implementación de la red neuronal U-Net se ha realizado con el *framework* de Pytorch.

### 4.1.1 Preprocesado de datos de U-Net

Para alimentar a la red neuronal U-Net, cada par de datos (imagen RGB y su correspondiente máscara de etiquetas) pasa los siguientes pasos de preprocesado: en primer lugar, las imágenes se cargan del *dataset* y se redimensionan a 512 x 256 (manteniendo la proporción 2:1 característica de Cityscapes), mientras que las máscaras se reescalan al mismo tamaño empleando interpolación por vecino más cercano *nereast neighbor* para preservar los valores originales de la máscara de etiqueta. Por último, la imagen se normaliza al rango [0, 1] dividiendo por 255.

La red U-Net recibe lotes de 6 imágenes, produciéndose así un total de 496 lotes, donde el último se compone de 5 imágenes.

### 4.1.2 Arquitectura e hiperparámetros de la implementación U-Net

La red U-Net diseñada para segmentación semántica de Cityscapes toma como entrada imágenes con las siguientes dimensiones 3x256x512. El *encoder* consta de cuatro bloques secuenciales de doble convolución 3x3 con activación ReLU, seguidos cada uno de ellos por una operación de reducción de dimensionalidad *max pooling* 2x2. Los canales de salida crecen de 64, 128, 256 a 512, y se añade una capa *dropout* para mejorar la generalización, aplicada tras el cuarto bloque. A continuación, el *middle block* de la arquitectura emplea otro bloque de doble convolución que expande la profundidad a 1024 canales e incluye una segunda capa *dropout*. El decodificador realiza cuatro pasos de *upsampling* mediante convoluciones transpuestas con *kernel* 2x2 y *stride* 2, reduciendo progresivamente los canales de 1024 a 512, 256, 128 y 64; en cada paso se concatena ("*skip connection*") la salida de la parte *downsampling* con la característica correspondiente del codificador y refina el resultado con un bloque de doble convolución. Finalmente, una convolución 1x1 proyecta los 64 canales finales a las 19 clases semánticas.

La arquitectura de la red U-Net mencionada se resume a continuación:

```
================================================
RESUMEN DEL MODELO U-NET
================================================
Layer (type)          Output Shape        Param #
================================================
Conv2d-1          [-1, 64, 256, 512]         1,792
ReLU-2            [-1, 64, 256, 512]         0
Conv2d-3          [-1, 64, 256, 512]         36,928
ReLU-4            [-1, 64, 256, 512]         0
MaxPool2d-5       [-1, 64, 128, 256]         0
Conv2d-6          [-1, 128, 128, 256]        73,856
ReLU-7            [-1, 128, 128, 256]        0
Conv2d-8          [-1, 128, 128, 256]        147,584
ReLU-9            [-1, 128, 128, 256]        0
MaxPool2d-10      [-1, 128, 64, 128]         0
Conv2d-11         [-1, 256, 64, 128]         295,168
ReLU-12           [-1, 256, 64, 128]         0
Conv2d-13         [-1, 256, 64, 128]         590,080
ReLU-14           [-1, 256, 64, 128]         0
MaxPool2d-15      [-1, 256, 32, 64]          0
Conv2d-16         [-1, 512, 32, 64]          1,180,160
ReLU-17           [-1, 512, 32, 64]          0
Conv2d-18         [-1, 512, 32, 64]          2,359,808
ReLU-19           [-1, 512, 32, 64]          0
Dropout-20        [-1, 512, 32, 64]          0
MaxPool2d-21      [-1, 512, 16, 32]          0
Conv2d-22         [-1, 1024, 16, 32]         4,719,616
ReLU-23           [-1, 1024, 16, 32]         0
Conv2d-24         [-1, 1024, 16, 32]         9,438,208
ReLU-25           [-1, 1024, 16, 32]         0
Dropout-26        [-1, 1024, 16, 32]         0
ConvTranspose2d-27     [-1, 512, 32, 64]     2,097,664
Conv2d-28         [-1, 512, 32, 64]          4,719,104
ReLU-29           [-1, 512, 32, 64]          0
Conv2d-30         [-1, 512, 32, 64]          2,359,808
ReLU-31           [-1, 512, 32, 64]          0
ConvTranspose2d-32     [-1, 256, 64, 128]    524,544
Conv2d-33         [-1, 256, 64, 128]         1,179,904
ReLU-34           [-1, 256, 64, 128]         0
Conv2d-35         [-1, 256, 64, 128]         590,080
ReLU-36           [-1, 256, 64, 128]         0
ConvTranspose2d-37     [-1, 128, 128, 256]   131,200
Conv2d-38         [-1, 128, 128, 256]        295,040
ReLU-39           [-1, 128, 128, 256]        0
Conv2d-40         [-1, 128, 128, 256]        147,584
ReLU-41           [-1, 128, 128, 256]        0
ConvTranspose2d-42     [-1, 64, 256, 512]    32,832
Conv2d-43         [-1, 64, 256, 512]         73,792
ReLU-44           [-1, 64, 256, 512]         0
Conv2d-45         [-1, 64, 256, 512]         36,928
ReLU-46           [-1, 64, 256, 512]         0
Conv2d-47         [-1, 19, 256, 512]         1,235
================================================
=============
Total params: 31,032,915
Trainable params: 31,032,915
Non-trainable params: 0
----------------------------------------------------------------
```

*Figura 37. Resumen de Pytorch de la arquitectura de U-Net*

Para el entrenamiento se emplea *Cross-Entropy Loss*, que ignora los píxeles marcados como 255 (la clase *background*), y el optimizador Adam con tasa de aprendizaje fija de $10^{-4}$ entrenando durante un máximo de 100 épocas.

### 4.1.3 Resultados de U-Net

Los resultados obtenidos con este modelo se analizan observando cómo converge la función de pérdida de la red junto a su desempeño con el dataset de validación, mediante el cálculo del mIoU que consiste en el IoU promedio de la máscara de segmentación devuelta por la red con la que debería ser.

A continuación, se muestra una gráfica con la convergencia de la función de pérdida de la red a lo largo de 100 épocas:

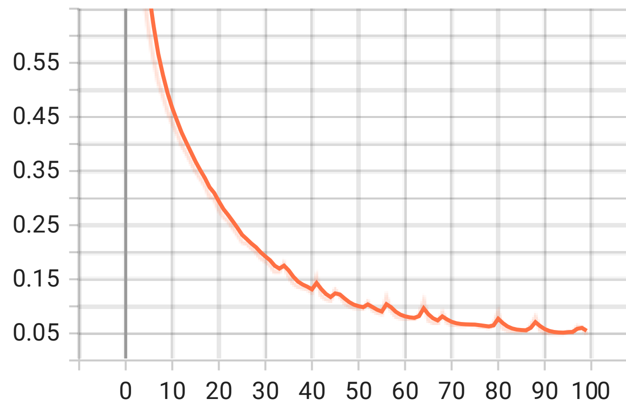



*Figura 38. Gráfica de la evolución de la función de pérdida de U-Net*

Como se observa en la gráfica la red va reduciendo su error a lo largo de las 100 épocas, indicativo de que se encuentra aprendiendo. Cuando alcanza las 80 épocas el error de la red se empieza a estabilizar, lo que sugiere una convergencia alcanzada en un valor aproximado a 0.05. Como aclaración para esta gráfica de resultados y las siguientes la sombra en ella indica que ha sido “suavizada” un 0.5 en tensorboard con el objetivo de conseguir mayor legibilidad.

Otra de las métricas que explica que tan bien aprende la red es la de *validation accuracy* que indica la proporción de píxeles clasificados correctamente respecto al total de píxeles. En la siguiente gráfica se observa la *validation accuracy* a lo largo de 100 épocas:

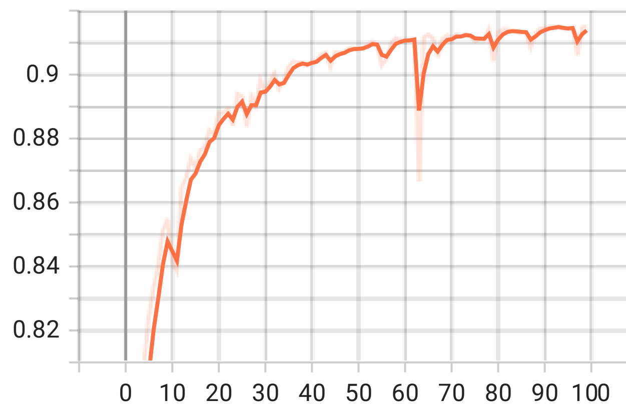



*Figura 39. Gráfica de la evolución de la validation accuracy de U-Net*

Como se observa en la gráfica existe una evolución positiva a lo largo de las épocas lo que es indicativo de una mejora en la predicción global de los píxeles. En este caso se comprueba que el modelo acaba acertando aproximadamente el 90% de los píxeles cuando llega al final de su entrenamiento.

Por último, una métrica algo más robusta que la *validation accuracy* es el mIoU (*Mean Intersection Over Union*) ya que a diferencia de la otra, esta da igual peso a todas las clases, y es que en el *dataset* de Cityscapes hay clases que predominan más en número de píxeles que otras, como por ejemplo, la clase carretera que contiene significativamente más píxeles en promedio por imagen que la clase persona. Es por ello por lo que esta métrica es muy útil para ver que tan bien predice el modelo las clases con independencia de su predominancia en el *dataset*. A continuación, se muestra una gráfica del mIoU sobre el *dataset* de validación a lo largo de las 100 épocas de entrenamiento:

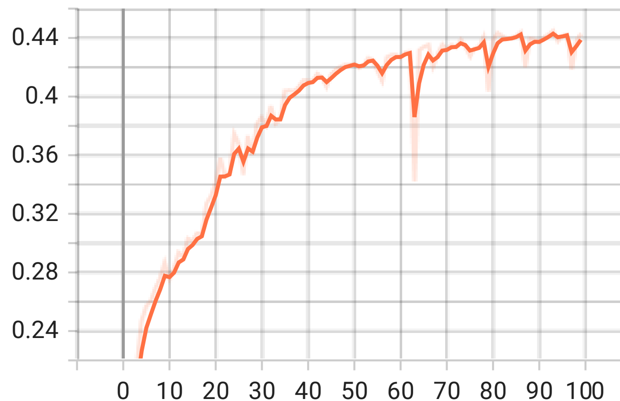



*Figura 40. Gráfica de la evolución del mIoU de U-Net*

En este caso una convergencia ascendente también es indicativo de la mejora de predicción de máscaras del modelo, indicándose que en promedio predice el 44% de los píxeles correctamente independientemente de su clase. Este criterio de promediar todas las clases por igual evita que las clases más abundantes (por ejemplo, "carretera") dominen la métrica y penaliza con el mismo peso a las clases más escasas. Este valor puede deberse a que U-Net al ser un modelo relativamente simple y el *dataset* bastante pequeño pueden existir clases que le cueste más predecir (las de menor aparición).

Los resultados proporcionados por el modelo en distintas etapas cogiendo imágenes del *dataset* de validación se muestran a continuación:

- En la época 5 la función objetivo toma el valor de 0.7347 obteniéndose las imágenes mostradas en la figura 41:

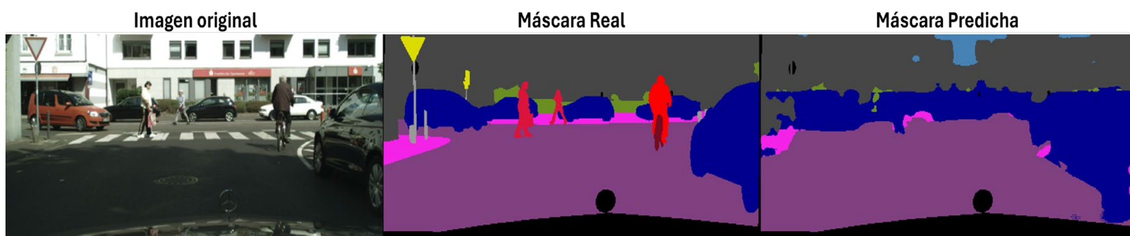



*Figura 41. Resultados de U-Net en la época 5*

- En la época 55 la función objetivo toma el valor de 0.09392 obteniéndose las imágenes mostradas en la figura 42:

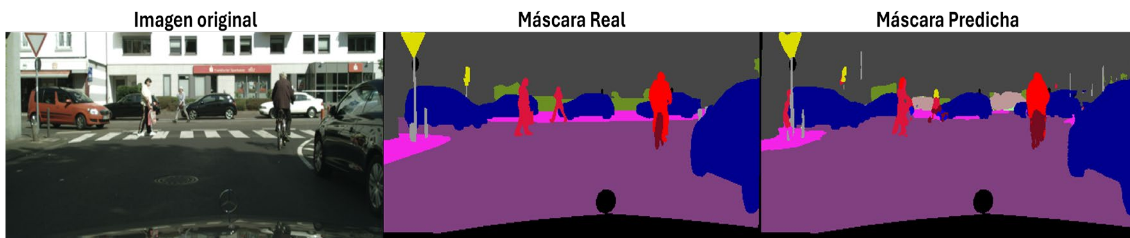



*Figura 42. Resultados de U-Net en la época 55*

- En la época 100 la función objetivo toma el valor de 0.04946 obteniéndose las imágenes mostradas en la figura 43:

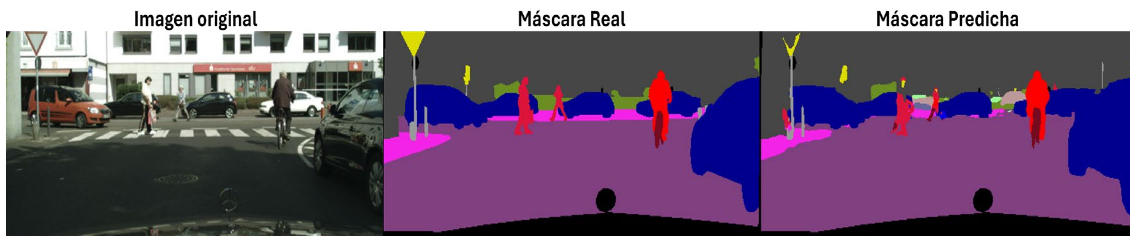



*Figura 43. Resultados de U-Net en la época 100*

Como se puede ver a medida que pasan las épocas las imágenes van mejorando. Parece que la de la época 54 es algo mejor que la de la época 100, esto en parte se puede deber a un sobreajuste del modelo que en detalles finos puede producir mayores errores con el *dataset* de validación, sin embargo, aunque en este caso se vea mejor la figura 42, como se ve en la gráfica de mIoU el valor es superior en la época 100 que en la 54 lo que indica que como tal el modelo mejora en el promedio de imágenes del dataset de validación a la hora de predecir píxeles con independencia de la clase.

## 4.2 Arquitectura DeepLabv3+

Toda la implementación de la red neuronal DeepLabv3+ se ha realizado con el *framework* de Pytorch.

### 4.2.1 Arquitectura DeepLabv3+ del artículo original

En este apartado se detalla la construcción y los resultados del modelo DeepLabv3+ basada en la implementación del artículo original.

#### 4.2.1.1 Preprocesado de datos de DeepLabv3+

El preprocesado realizado para esta implementación de DeepLabv3+ es el mismo que el empleado en U-Net, lo único que el tamaño de lote cambia, en este caso es de 48 imágenes, dando un total de 62 lotes por época de los cuales el último contiene 47 imágenes.

#### 4.2.1.2 Arquitectura e hiperparámetros de la implementación DeepLabv3+

Este modelo se compone de un *backbone* basado en una versión modificada de Xception como mencionan en el artículo original, seguido por un módulo ASPP y un decodificador que combina características de alto y bajo nivel para mejorar la precisión espacial.

El *backbone* Xception está estructurado en tres flujos principales:

- ***Entry Flow***: Comienza con dos convoluciones estándar (3x3) que aumentan progresivamente los canales (de 3 a 64), seguidas por tres bloques consistentes en dos ramas paralelas, una con una convolución 1x1 y la otra rama con tres convoluciones separables *atrous* 3x3 que reducen la resolución

espacial hasta un total de 728 canales (ambas ramas se concatenan al final de cada bloque como una *skip connection*). El último bloque ajusta su *stride* según el *output stride* elegido (que en este caso es 16). Todas las capas cuentan con ReLU de función de activación y BatchNorm.

- ***Middle Flow***: Compuesto por 16 bloques como los de *entry Flow*, pero sin la convolución 1x1, repetidos, con dilatación igual a 1, y *skip connections* de tipo identidad, manteniendo 728 canales.
- ***Exit Flow***: Incluye un bloque igual que el de *entry Flow*, pero con dilatación 1 y salida a 1024 canales, seguido por tres convoluciones separables adicionales que expanden los canales a 1536 y 2048 respectivamente, cada una acompañada de BatchNorm y ReLU.

El módulo ASPP procesa las características extraídas por el codificador mediante cinco ramas paralelas: una convolución 1x1 sin dilatación, tres convoluciones 3x3 con diferentes tasas de dilatación (1, 6, 12, 18) para el *output stride* 16, y una rama de *Global Average Pooling* 1x1 seguida de una capa de interpolación bilineal. Las salidas de estas ramas se concatenan y pasan por una convolución final 1x1 con BatchNorm, ReLU y *dropout* p=0.5.

El decodificador integra características de alto nivel provenientes de ASPP con características de bajo nivel extraídas del codificador (salida de la primera segunda convolución del *entry flow*): las características intermedias del codificador (64 canales) se proyectan a 48 mediante una convolución 1x1. A continuación, las características del ASPP (256 canales) se interpolan al tamaño espacial de las características de bajo nivel y se concatenan (304 canales). Esta combinación se procesa mediante dos bloques convolucionales 3x3 con BatchNorm, ReLU y *dropout* (0.5 y 0.1). Finalmente, una convolución 1x1 proyecta a 19 clases semánticas, y el resultado se interpola al tamaño original 256x512.

Para el entrenamiento se utiliza la misma función de pérdida que la de U-Net, *Cross-Entropy Loss*. El optimizador es SGD con momento 0.9, *weight decay* $5x10^{-4}$, y una política de *learning rate* polinómica (PolyLR) con tasas diferenciadas para componentes clave: 0.001 para el *backbone* y 0.01 para ASPP, decodificador y clasificador. La duración máxima de entrenamiento es de 200 épocas.

La arquitectura del modelo Deeplabv3+ presentada, basada en el artículo original se muestra a continuación (se han eliminado las capas BatchNorm y ReLU para simplificar, por ello si se hace la suma de parámetros no da igual al total de parámetros):

RESUMEN DEL MODELO DeepLabv3+ ORIGINAL

| Layer (type) | Output Shape | Param # |
|---|---|---|
| Conv2d-1 | [-1, 32, 128, 256] | 864 |
| Conv2d-4 | [-1, 64, 128, 256] | 18,432 |
| Conv2d-8 | [-1, 64, 128, 256] | 576 |
| Conv2d-9 | [-1, 128, 128, 256] | 8,192 |
| SeparableConv2d-10 | [-1, 128, 128, 256] | 0 |
| Conv2d-13 | [-1, 128, 128, 256] | 1,152 |
| Conv2d-14 | [-1, 128, 128, 256] | 16,384 |
| SeparableConv2d-15 | [-1, 128, 128, 256] | 0 |
| Conv2d-18 | [-1, 128, 64, 128] | 1,152 |
| Conv2d-19 | [-1, 128, 64, 128] | 16,384 |
| SeparableConv2d-20 | [-1, 128, 64, 128] | 0 |
| Conv2d-22 | [-1, 128, 64, 128] | 8,192 |
| XceptionBlock-24 | [-1, 128, 64, 128] | 0 |
| Conv2d-26 | [-1, 128, 64, 128] | 1,152 |
| Conv2d-27 | [-1, 256, 64, 128] | 32,768 |
| SeparableConv2d-28 | [-1, 256, 64, 128] | 0 |
| Conv2d-31 | [-1, 256, 64, 128] | 2,304 |
| Conv2d-32 | [-1, 256, 64, 128] | 65,536 |
| SeparableConv2d-33 | [-1, 256, 64, 128] | 0 |
| Conv2d-36 | [-1, 256, 32, 64] | 2,304 |
| Conv2d-37 | [-1, 256, 32, 64] | 65,536 |
| SeparableConv2d-38 | [-1, 256, 32, 64] | 0 |
| Conv2d-40 | [-1, 256, 32, 64] | 32,768 |
| XceptionBlock-42 | [-1, 256, 32, 64] | 0 |
| Conv2d-44 | [-1, 256, 32, 64] | 2,304 |
| Conv2d-45 | [-1, 728, 32, 64] | 186,368 |
| SeparableConv2d-46 | [-1, 728, 32, 64] | 0 |
| Conv2d-49 | [-1, 728, 32, 64] | 6,552 |
| Conv2d-50 | [-1, 728, 32, 64] | 529,984 |
| SeparableConv2d-51 | [-1, 728, 32, 64] | 0 |
| Conv2d-54 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-55 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-56 | [-1, 728, 16, 32] | 0 |
| Conv2d-58 | [-1, 728, 16, 32] | 186,368 |
| XceptionBlock-60 | [-1, 728, 16, 32] | 0 |
| Conv2d-62 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-63 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-64 | [-1, 728, 16, 32] | 0 |
| Conv2d-67 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-68 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-69 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-72 | [-1, 728, 16, 32] | 0 |
| Conv2d-74 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-75 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-76 | [-1, 728, 16, 32] | 0 |
| Conv2d-79 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-80 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-81 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-84 | [-1, 728, 16, 32] | 0 |
| Conv2d-86 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-87 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-88 | [-1, 728, 16, 32] | 0 |
| Conv2d-91 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-92 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-93 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-96 | [-1, 728, 16, 32] | 0 |
| Conv2d-98 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-99 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-100 | [-1, 728, 16, 32] | 0 |
| Conv2d-103 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-104 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-105 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-108 | [-1, 728, 16, 32] | 0 |
| Conv2d-110 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-111 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-112 | [-1, 728, 16, 32] | 0 |
| Conv2d-115 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-116 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-117 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-120 | [-1, 728, 16, 32] | 0 |
| Conv2d-122 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-123 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-124 | [-1, 728, 16, 32] | 0 |
| Conv2d-127 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-128 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-129 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-132 | [-1, 728, 16, 32] | 0 |
| Conv2d-134 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-135 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-136 | [-1, 728, 16, 32] | 0 |
| Conv2d-139 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-140 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-141 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-144 | [-1, 728, 16, 32] | 0 |
| Conv2d-146 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-147 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-148 | [-1, 728, 16, 32] | 0 |
| Conv2d-151 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-152 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-153 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-156 | [-1, 728, 16, 32] | 0 |
| Conv2d-158 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-159 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-160 | [-1, 728, 16, 32] | 0 |
| Conv2d-163 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-164 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-165 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-168 | [-1, 728, 16, 32] | 0 |
| Conv2d-170 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-171 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-172 | [-1, 728, 16, 32] | 0 |
| Conv2d-175 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-176 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-177 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-180 | [-1, 728, 16, 32] | 0 |
| Conv2d-182 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-183 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-184 | [-1, 728, 16, 32] | 0 |
| Conv2d-187 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-188 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-189 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-192 | [-1, 728, 16, 32] | 0 |
| Conv2d-194 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-195 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-196 | [-1, 728, 16, 32] | 0 |
| Conv2d-199 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-200 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-201 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-204 | [-1, 728, 16, 32] | 0 |
| Conv2d-206 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-207 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-208 | [-1, 728, 16, 32] | 0 |
| Conv2d-211 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-212 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-213 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-216 | [-1, 728, 16, 32] | 0 |
| Conv2d-218 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-219 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-220 | [-1, 728, 16, 32] | 0 |
| Conv2d-223 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-224 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-225 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-228 | [-1, 728, 16, 32] | 0 |
| Conv2d-230 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-231 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-232 | [-1, 728, 16, 32] | 0 |
| Conv2d-235 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-236 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-237 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-240 | [-1, 728, 16, 32] | 0 |
| Conv2d-242 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-243 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-244 | [-1, 728, 16, 32] | 0 |
| Conv2d-247 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-248 | [-1, 728, 16, 32] | 529,984 |
| SeparableConv2d-249 | [-1, 728, 16, 32] | 0 |
| XceptionBlock-252 | [-1, 728, 16, 32] | 0 |
| Conv2d-254 | [-1, 728, 16, 32] | 6,552 |
| Conv2d-255 | [-1, 1024, 16, 32] | 745,472 |
| SeparableConv2d-256 | [-1, 1024, 16, 32] | 0 |
| Conv2d-259 | [-1, 1024, 16, 32] | 9,216 |
| Conv2d-260 | [-1, 1024, 16, 32] | 1,048,576 |
| SeparableConv2d-261 | [-1, 1024, 16, 32] | 0 |
| Conv2d-264 | [-1, 1024, 16, 32] | 745,472 |
| XceptionBlock-266 | [-1, 1024, 16, 32] | 0 |
| Conv2d-267 | [-1, 1024, 16, 32] | 9,216 |
| Conv2d-268 | [-1, 1536, 16, 32] | 1,572,864 |
| SeparableConv2d-269 | [-1, 1536, 16, 32] | 0 |
| Conv2d-272 | [-1, 1536, 16, 32] | 13,824 |
| Conv2d-273 | [-1, 2048, 16, 32] | 3,145,728 |
| SeparableConv2d-274 | [-1, 2048, 16, 32] | 0 |
| Conv2d-277 | [-1, 2048, 16, 32] | 18,432 |
| Conv2d-278 | [-1, 2048, 16, 32] | 4,194,304 |
| SeparableConv2d-279 | [-1, 2048, 16, 32] | 0 |
| XceptionEncoder-282 | [[-1, 2048, 16, 32], [-1, 64, 128, 256]] | 0 |
| Conv2d-283 | [-1, 256, 16, 32] | 524,288 |
| Conv2d-286 | [-1, 2048, 16, 32] | 18,432 |
| Conv2d-287 | [-1, 256, 16, 32] | 524,288 |
| SeparableConv2d-288 | [-1, 256, 16, 32] | 0 |
| Conv2d-291 | [-1, 2048, 16, 32] | 18,432 |
| Conv2d-292 | [-1, 256, 16, 32] | 524,288 |
| SeparableConv2d-293 | [-1, 256, 16, 32] | 0 |
| Conv2d-296 | [-1, 2048, 16, 32] | 18,432 |
| Conv2d-297 | [-1, 256, 16, 32] | 524,288 |
| SeparableConv2d-298 | [-1, 256, 16, 32] | 0 |
| AdaptiveAvgPool2d-301 | [-1, 2048, 1, 1] | 0 |
| Conv2d-302 | [-1, 256, 1, 1] | 524,288 |
| Conv2d-305 | [-1, 256, 16, 32] | 327,680 |
| Dropout-308 | [-1, 256, 16, 32] | 0 |
| ASPP-309 | [-1, 256, 16, 32] | 0 |
| Conv2d-310 | [-1, 48, 128, 256] | 3,072 |
| Conv2d-313 | [-1, 256, 128, 256] | 700,416 |
| Dropout-316 | [-1, 256, 128, 256] | 0 |
| Conv2d-317 | [-1, 256, 128, 256] | 589,824 |
| Dropout-320 | [-1, 256, 128, 256] | 0 |
| Conv2d-321 | [-1, 19, 128, 256] | 4,883 |

Total params: 34,780,507
Trainable params: 34,780,507
Non-trainable params: 0

*Figura 44. Resumen de Pytorch de la arquitectura de DeepLabv3+ original*

#### 4.2.1.3 Resultados de DeepLabv3+

Los resultados obtenidos con la implementación del artículo original sin preetrenamiento se analizan partiendo de la convergencia de las gráficas de la función de pérdida, de la *validation accuracy* y el mIoU del *dataset* de validación de igual manera que en la red U-Net.

A continuación, se muestra una gráfica con la convergencia de la función de pérdida de la red a lo largo de 200 épocas:

*Figura 45. Gráfica de la evolución de la función de pérdida de U-Net*

Como se observa en la gráfica la red va reduciendo su error a lo largo de las 200 épocas, por lo que se concluye que la red aprende con los datos de entrenamiento. El valor del error en este caso se comienza a estabilizar sobre 0.15.

En la siguiente gráfica se observa la *validation accuracy* a lo largo de 200 épocas de entrenamiento:

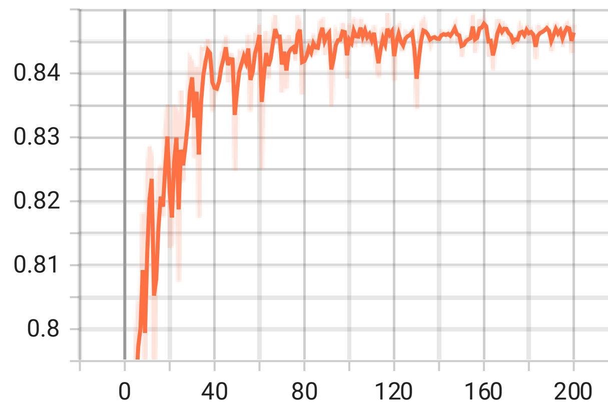



*Figura 46. Gráfica de la evolución de la validation accuracy de U-Net*

En esta gráfica se observa como el modelo va obteniendo mejor precisión a lo largo del entrenamiento hasta estabilizarse en valores cercanos a 0.85 lo que indica que consigue predecir bien el 85% de los píxeles del *dataset* de validación.

En cuanto al mIoU obtenido con este modelo, a continuación se presentan los resultados:

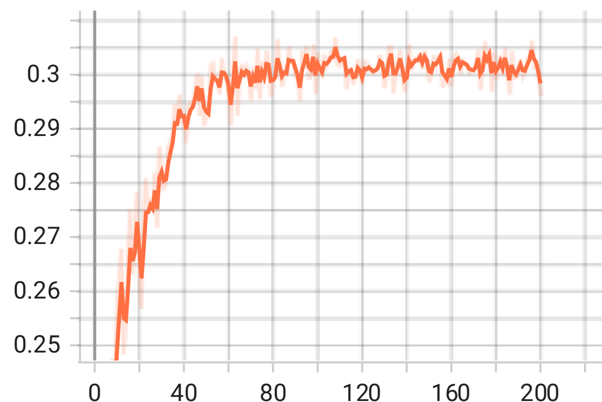



*Figura 47. Gráfica de la evolución del mIoU de U-Net*

Con este modelo el mIoU es bastante inferior que los resultados obtenidos en U-Net, llegándose a valores bastante bajos de en torno a 0.3. Puede haber varias explicaciones para este resultado, una que se plantea es que este modelo es mucho más complejo y consta de muchas más capas que la red U-Net y que al ser este *dataset* de entrenamiento muy pequeño puede que no esté siendo capaz de generalizar lo suficiente y de aprender correctamente los patrones de distintas clases, un aspecto que podría ser clave para un mejor resultado es el preentrenamiento del *backbone* del modelo sobre *datasets* como ImageNet como hacen en el artículo original.

Los resultados proporcionados por el modelo en distintas etapas cogiendo imágenes del *dataset* de validación se muestran a continuación:

- En la época 1 la función objetivo toma el valor de 1.208 obteniéndose las imágenes mostradas en la figura 48:

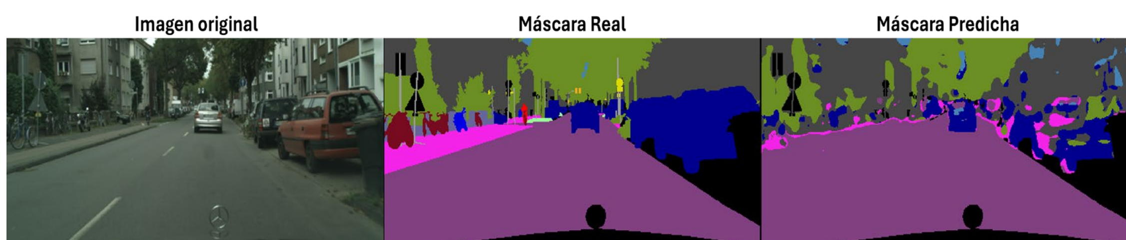



*Figura 48. Resultados de DeepLabv3+ en la época 1*

- En la época 100 la función objetivo toma el valor de 0.1911 obteniéndose las imágenes mostradas en la figura 49:

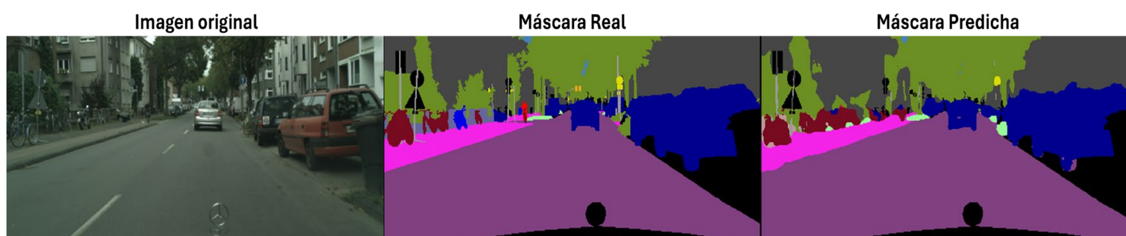



*Figura 49. Resultados de DeepLabv3+ en la época 100*

- En la época 200 la función objetivo toma el valor de 0.1381 obteniéndose las imágenes mostradas en la figura *50*:

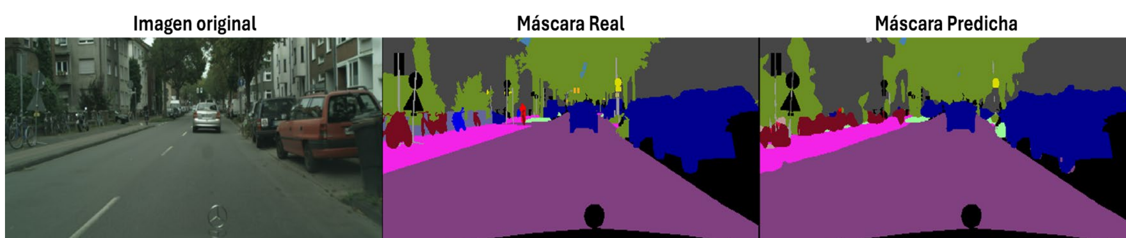



*Figura 50. Resultados de DeepLabv3+ en la época 200*

Se observa que a medida que pasan las épocas las imágenes van mejorando lo cual concuerda con las gráficas anteriormente presentadas. A pesar de que el mIoU es bastante bajo pareciera con estos ejemplos las máscaras de predicción son decentes (aunque se aprecian errores) y es que el mIoU realiza la media sobre todas las imágenes del *dataset* de validación, es decir, que aunque en este caso parezcan muy decentes las máscaras puede haber casos donde el modelo falle bastante.

### 4.2.2 Arquitectura modificada

En este apartado se detalla la construcción y los resultados obtenidos utilizando una implementación del modelo DeepLabv3+ sacada de un repositorio de github [65] para hacer la comparativa con la implementación realizada.

#### 4.2.2.1 Preprocesado de datos de DeepLabv3+

Para entrenar a esta red neuronal se realizan una serie de pasos distintos a los de los apartados previos ya que se observa en este caso particular que este modelo de DeepLabv3+ que consta de un *backbone* reducido mejora con este preprocesado. En este caso durante la fase de entrenamiento, se aplican transformaciones de aumento de datos para mejorar la generalización del

modelo: se realiza un recorte aleatorio a un tamaño de 768x768 píxeles, se aplican variaciones de color en brillo, contraste y saturación con una intensidad de 0.5, y se emplea un volteo horizontal aleatorio con una probabilidad de 0.5.

A continuación, tanto las imágenes como las máscaras se convierten en tensores: las imágenes RGB se transforman en tensores con valores en el rango [0.0, 1.0] y forma (3x768x768), mientras que las máscaras se convierten en tensores con la misma resolución espacial. Las imágenes se normalizan utilizando la media y desviación estándar utilizada para entrenamientos de clasificación de imágenes en el *dataset* de ImageNet, con valores mean = [0.485, 0.456, 0.406] y std = [0.229, 0.224, 0.225].

Los datos procesados se agrupan en lotes de 16 muestras durante el entrenamiento. Dado que el conjunto de entrenamiento consta de aproximadamente 2975 imágenes y se utiliza la opción eliminar el último lote si no está completo, se generan 185 lotes completos para recorrer el *dataset*.

#### 4.2.2.2 Arquitectura e hiperparámetros de la implementación DeepLabv3+

La arquitectura de la implementación de github DeepLabv3+ utiliza como entrada imágenes RGB a un tamaño de 3x768x768 como se ha mencionado. El modelo está compuesto por un *backbone* Xception (más reducido que la del artículo de DeepLabv3+, concretamente como el del Xception original), seguido por un módulo ASPP y un decodificador de igual forma que en la otra implementación. El *output stride* del *backbone* está configurado en 16.

El *backbone* Xception se organiza en los siguientes tres flujos:

- ***Entry Flow***: comienza con dos convoluciones estándar (3x3) con *strides* de 2 y 1 respectivamente, que incrementan los canales de 3 a 64. Cada convolución está seguida de una capa de BatchNorm y una función de activación ReLU. A continuación, se aplican tres bloques Xception compuestos por convoluciones separables *atrous*, cada una acompañada de BatchNorm y ReLU. Estos bloques reducen progresivamente la resolución espacial y elevan los canales hasta un total de 728.
- ***Middle Flow***: consiste en ocho bloques XceptionBlock repetidos, a diferencia de los 16 de la implementación del artículo original, todos con dilatación igual a 1 y conexiones residuales tipo identidad, manteniendo constante el número de canales en 728.
- ***Exit Flow***: incluye un bloque Xception que expande los canales de 728 a 1024, seguido por dos convoluciones separables adicionales que aumentan los canales a 1536 y posteriormente a 2048. Cada una de estas capas está acompañada por BatchNorm y ReLU.

El resto de arquitectura es igual que la otra implementación.

En este caso en el entrenamiento se realizan un total de 11200 iteraciones en el caso de este modelo sin preetrenamiento y de 30 000 iteraciones con preetrenamiento. Se pruebas los mismos hiperparámetros que en la otra arquitectura.

#### 4.2.2.3 Resultados de DeepLabv3+ sin preetrenamiento

A lo largo de este apartado se muestran los resultados obtenidos por el modelo de DeepLabv3+ de github sin preetrenamiento. En este caso se debe mencionar que se habla de iteraciones (*steps*) ya que el código de github no imprime las épocas en tensorboard, pero aproximadamente 180 iteraciones equivalen a 1 época, por lo que se evalúa el modelo hasta la época 62 que es donde se empieza a apreciar el punto de convergencia de la red.

Lo primero a analizar es la función de pérdida de la red a lo largo de estas 11200 iteraciones. A continuación, se muestra la gráfica de la función de pérdida:

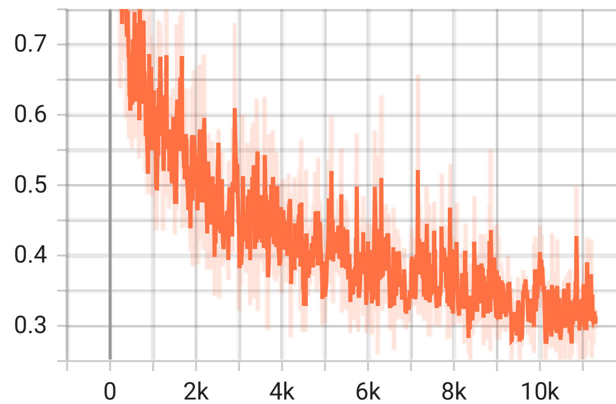



*Figura 52. Gráfica de la evolución de la función de pérdida de DeepLabv3+*

En esta gráfica se observa como la red va reduciendo su error a lo largo de las 11200 iteraciones, por lo que se concluye que la red está aprendiendo. El valor del error en este caso se comienza a estabilizar sobre 0.3.

En la siguiente gráfica se muestra la *validation accuracy* a lo largo de 11200 iteraciones de entrenamiento:

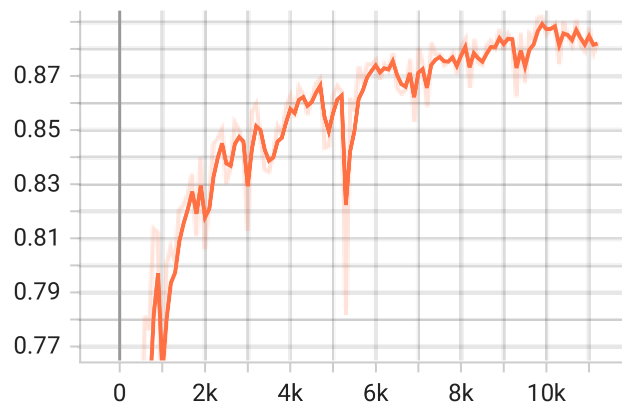



*Figura 53. Gráfica de la evolución de la validation accuracy de DeepLabv3+*

La gráfica de *validation accuracy* muestra una tendencia creciente que se estabiliza en valores próximos a 0.89 de precisión, es decir, un 89% de píxeles correctamente predichos.

En cuanto al mIoU obtenido con este modelo, a continuación se presentan los resultados:

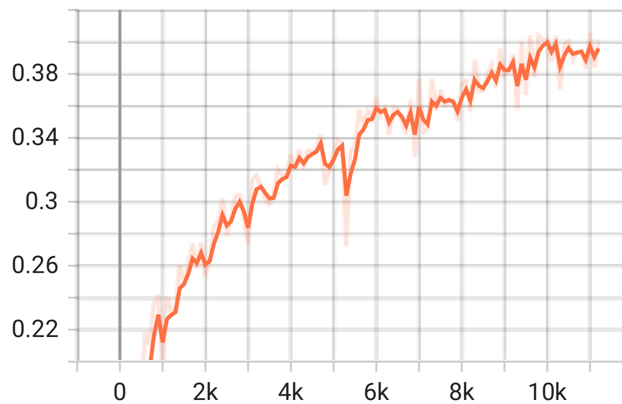


*Figura 54. Gráfica de la evolución del mIoU de DeepLabv3+*

Con este modelo el mIoU alcanza valores cercanos a 0.4 similar al modelo de U-Net. En cuanto al por qué en la gráfica de la función de pérdida de este modelo se observan valores de error significativamente superiores a los de la otra implementación presentada de DeepLabv3+, esto se puede deber a que el otro modelo parece haberse adaptado mejor al *dataset* de entrenamiento (tiene un menor *train loss*), pero que, sin embargo, por la arquitectura del *backbone* de este modelo, algo más simple, y por las transformaciones de las imágenes en el preprocesado ,este modelo se adapta mejor al *dataset* de validación (es capaz de generalizar más, que es lo que se busca).

Los resultados proporcionados por el modelo en las distintas iteraciones cogiendo imágenes del *dataset* de validación se muestran a continuación:

- En la iteración 200 la función objetivo toma el valor de 0.817 obteniéndose las imágenes mostradas en la figura 55:

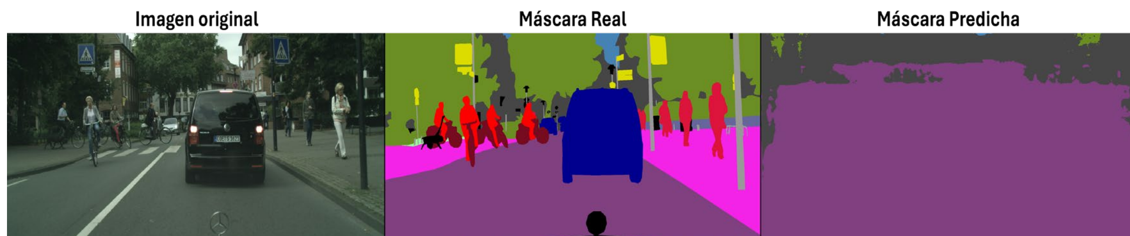


*Figura 55. Resultados de DeepLabv3+ en la iteración 200*

- En la iteración 5700 la función objetivo toma el valor de 0.4089 obteniéndose las imágenes mostradas en la figura 56:

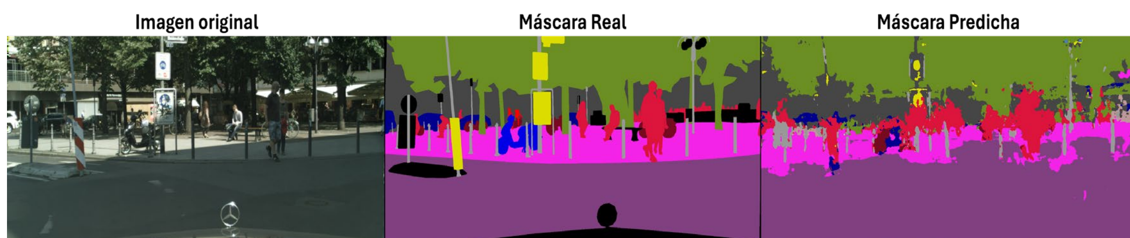


*Figura 56. Resultados de DeepLabv3+ en la iteración 5700*

- En la iteración 11200 la función objetivo toma el valor de 0.2556 obteniéndose las imágenes mostradas en la figura 57:

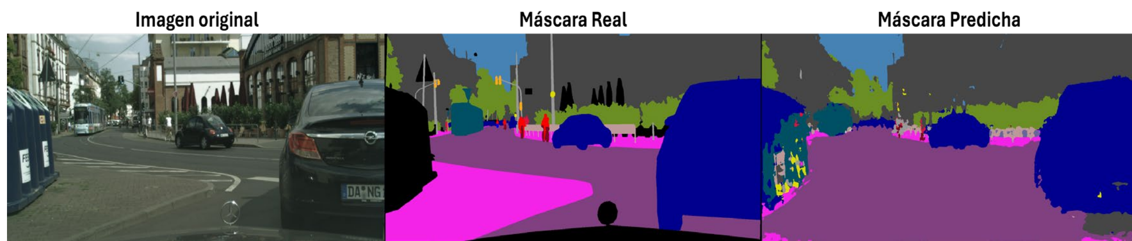



*Figura 57. Resultados de DeepLabv3+ en la iteración 11200*

Las máscaras de segmentación mejoran a medida que avanza el entrenamiento lo cual permite concluir un entrenamiento correcto. Sin embargo, como se comprueba en el resultado a pesar de que el mIoU es superior al del modelo anterior este parece dar peores máscaras lo cual puede deberse a que se está especializando quizás en detalles más finos o que no se aprecian tanto visualmente debido al preprocesado realizado, lo cual hace que el mIoU aumente, pero que el resultado visual parezca inferior. En este caso por cómo funciona el código de este modelo las máscaras generadas contienen valores para los píxeles *background* (los negros) como el capó del coche que toma las imágenes, aunque estos no se tienen en cuenta a la hora de calcular las métricas presentadas.

#### 4.2.2.4 Resultados de DeepLabv3+ con preetrenamiento

En este apartado se detallan los resultados obtenidos con esta versión de github, pero utilizando el *backbone* Xception mencionado preentrenado con el *dataset* ImageNet.

A continuación, se muestra la gráfica de la función de pérdida a lo largo de las 30000 iteraciones:

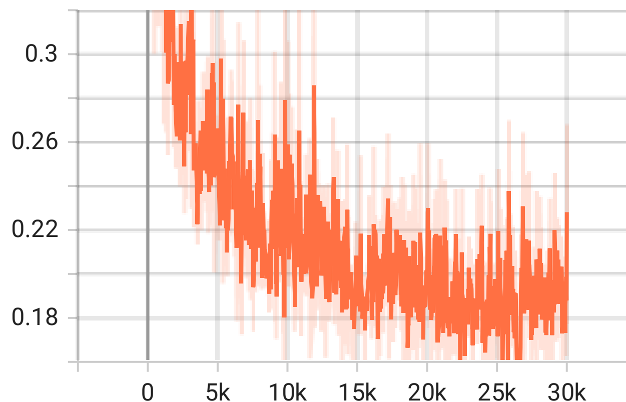



*Figura 58. Gráfica de la evolución de la función de pérdida de DeepLabv3+*

La gráfica muestra una evolución del error con tendencia decreciente llegando a valores próximos a 0.16.

En la siguiente gráfica se muestra la *validation accuracy* a lo largo de las 30000 iteraciones de entrenamiento:

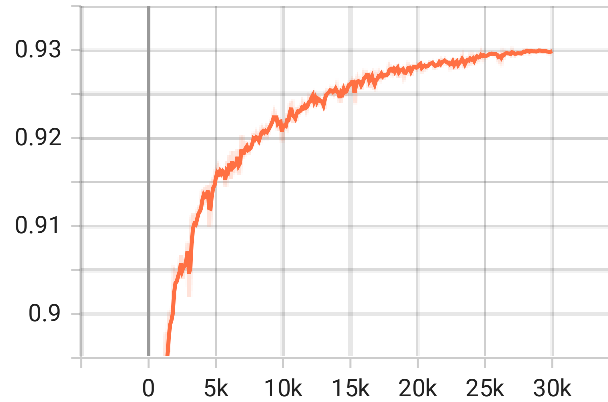



*Figura 59. Gráfica de la evolución de la validation accuracy de DeepLabv3+*

La gráfica de *validation accuracy* muestra una tendencia creciente que se estabiliza en valores próximos a 0.93 de precisión, es decir, un 93% de píxeles correctamente predichos.

En cuanto al mIoU obtenido con este modelo, a continuación se presentan los resultados:

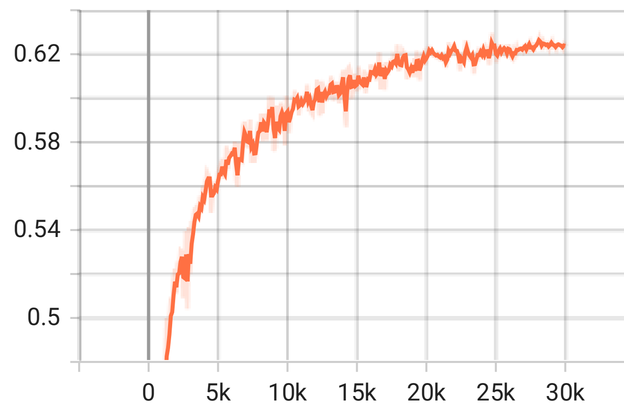



*Figura 60. Gráfica de la evolución del mIoU de DeepLabv3+*

En este caso se aprecia considerablemente el haber utilizado un *backbone* preentrenado con ImageNet ya que se alcanzan valores cercanos a 0.62. Esto puede concordar con la hipótesis planteada de que la implementación de DeepLabv3+ basada en el artículo original no llegue a valores tan altos al no haber sido preentrenado el *backbone* como mencionan en el artículo y al ser encima un modelo más complejo que este de github.

Los resultados proporcionados, por el modelo de github preentrenado, en las distintas iteraciones cogiendo imágenes del *dataset* de validación se muestran a continuación:

- En la iteración 100 la función objetivo toma el valor aproximado de 0.5713 obteniéndose las imágenes mostradas en la figura 61:

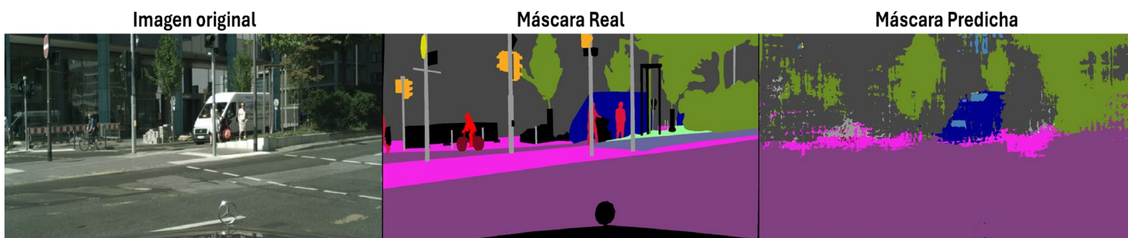



*Figura 61. Resultados de DeepLabv3+ en la iteración 100*

- En la iteración 15000 la función objetivo toma el valor de aproximadamente 0.1844 obteniéndose las imágenes mostradas en la figura *62*:

*Figura 62. Resultados de DeepLabv3+ en la iteración 15000*

- En la iteración 30000 la función objetivo toma el valor de 0.1774 (se toma el anterior valor, el 29980, porque justo en 30000 asciende repentinamente) obteniéndose las imágenes mostradas en la figura 63:

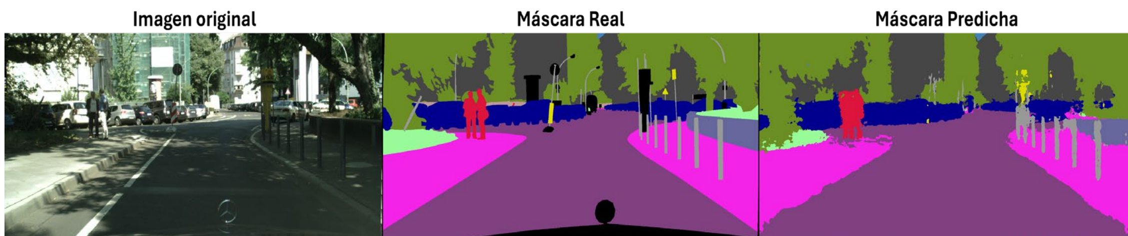



*Figura 63. Resultados de DeepLabv3+ en la iteración 30000*

De nuevo como se mencionó anteriormente se obtienen unos buenos resultados con este modelo debido a que ha sido preentrenado con el *dataset* ImageNet. Cabe destacar que parece que por el preprocesado realizado, aunque se mejore en general las máscaras, lleva a la red a generar zonas con trazados menos "suaves" que las otras implementaciones de U-Net y del modelo DeepLabv3+ basado en el artículo original.

## 4.3 Arquitectura Mask R-CNN

Toda la implementación de la red neuronal Mask R-CNN se ha realizado con el *framework* de Tensorflow.

En este apartado se detalla el cómo se construye el modelo Mask R-CNN utilizado. Modelo que ha sido sacado de un repositorio de github público [66], aunque la versión utilizada es una modificación de este repositorio para que sea compatible con Tensorflow 2.14 [67] y, por tanto, con las gráficas A100 de Nvidia utilizadas en Google Colab. También se muestran los resultados obtenidos con dicha implementación.

### 4.3.1 Preprocesado de datos de Mask R-CNN

El preprocesado de datos de la red neuronal Mask R-CNN funciona de la siguiente manera, cada par de datos (imagen RGB y sus correspondientes

anotaciones JSON) pasa los siguientes pasos de preprocesado: en primer lugar, las imágenes se cargan del *dataset* Cityscapes y se redimensionan a 1024x1024; simultáneamente, las máscaras binarias se generan a partir de los polígonos en los archivos JSON (rasterizando en resolución original 2048x1024, solo para las 8 clases de instancia) y luego se redimensionan a 1024x1024 con el mismo factor de escala. Por último, las imágenes se normalizan restando la media por canal [123.7, 116.8, 103.9] (sin dividir por desviación estándar), mientras que de las máscaras se extraen automáticamente los *bounding boxes*.

La red Mask R-CNN recibe lotes de 5 imágenes, por tanto, se procesan 595 lotes para completar todo el *dataset*.

### 4.3.2 Arquitectura e hiperparámetros de la implementación Mask R-CNN

La arquitectura de esta implementación de Mask R-CNN procesa imágenes RGB de tamaño 1024x1024x3. El modelo se estructura sobre un *backbone* ResNet-101, una FPN, una RPN y tres cabezas especializadas para clasificación, regresión de cajas y segmentación de máscaras. El flujo completo comienza tomando la imagen de entrada y extrayendo mapas de características a múltiples escalas, que luego alimentan tanto la RPN como las cabezas finales para obtener detecciones y máscaras refinadas.

El *backbone* ResNet-101 está organizado en cinco etapas sucesivas que reducen progresivamente la resolución espacial mientras aumentan la profundidad de los canales. En C1 hay una convolución 7x7 con *stride* 2 seguida de BatchNorm, ReLU y una capa MaxPool 3x3 con *stride* 2, lo que da un mapa de 64 canales. En C2 hay un ConvBlock seguido de dos IdentityBlocks con filtros [64, 64, 256] lo que produce un mapa de 256 canales. En C3 hay un ConvBlock y tres IdentityBlocks con filtros [128, 128, 512], resultando en 512 canales. C4 consta de un ConvBlock seguido de veintidós IdentityBlocks (para ResNet-101) con filtros [256, 256, 1024], produciendo 1024 canales. Finalmente, C5 contiene un ConvBlock y dos IdentityBlocks con filtros [512, 512, 2048], lo que resulta en 2048 canales.

La FPN aprovecha las salidas C2 a C5 del *backbone* para generar mapas P2 a P5, todos con 256 canales, y añade además P6 para la RPN. Primero, P5 se obtiene aplicando una convolución 1x1 a C5, reduciendo sus 2048 canales a 256 seguida de otra convolución 3x3. Luego, P4 se forma sumando el *upsampling* x2 de P5 con la proyección 1x1 de C4, seguido de una convolución 3x3. De igual manera se construyen P3 y P2, usando los mapas C3 y C2 respectivamente, cada uno fusionado con el *upsampling* x2 del nivel inmediatamente superior. Para P6, se aplica una capa *max pooling* 2x2 sobre P5, ofreciendo así anclas de mayor escala para la RPN.

La RPN opera simultáneamente sobre los mapas [P2, P3, P4, P5, P6]. En cada nivel, se utiliza una convolución 3x3 con 512 filtros cuyo *stride* se ajusta según la resolución de la pirámide. A continuación, se definen dos cabezas paralelas: una de clasificación que produce dos puntajes por anchor (*foreground* o *background*) y otra de regresión que genera cuatro valores de refinamiento de coordenadas por anchor. Los anchor se diseñan con escalas [32, 64, 128, 256, 512] y ratios [0.5, 1, 2], produciendo 15 anclas por posición espacial. Durante el entrenamiento, la RPN emplea *Non-Maximum Suppression* con umbral 0.7 y selecciona las 2000 mejores propuestas para alimentar las cabezas superiores.

Las tres cabezas especializadas procesan las ROIs generadas por la RPN mediante *Pyramid* ROI *Align* que realiza el alineamiento de las ROIs:

- La cabeza de clasificación usa dos capas *fully-connected* de 1024 neuronas cada una, seguidas de softmax para asignar probabilidades a cada clase (9 clases en Cityscapes).
- La cabeza de regresión de *bounding boxes* comparte la misma estructura de dos *fully-connected* de 1024, pero su salida es de tamaño nº clases x 4 para ajustar coordenadas específicas por clase.
- La cabeza de segmentación de máscaras consta de cuatro convoluciones 3x3 con 256 filtros, cada una seguida de BatchNorm y ReLU, luego una convolución transpuesta 2x2 (*stride* 2) y finalmente una convolución 1x1 con nº clases filtros y activación sigmoide, produciendo máscaras de 28x28.

Para el entrenamiento, se combinan varias funciones de pérdida: la RPN emplea *Cross-Entropy* para clasificación de *anchors* y *Smooth-L1* para regresión de coordenadas; la detección final utiliza *Sparse Categorical Cross-Entropy* para clasificación de objetos y *Smooth-L1* para el refinamiento de cajas; la segmentación de máscaras se optimiza con B*inary Cross-Entropy* entre máscaras predichas y *ground truth*. El optimizador es SGD con momento 0.9, *learning rate* inicial de 0.001 para las cabezas y 0.0001 (0.1x0.001) para el conjunto completo, con *gradient clipping* en norma 5.0. El entrenamiento se realiza en dos fases: primero se actualizan exclusivamente las cabezas durante 50 épocas y posteriormente se hace *fine-tuning* de toda la red durante otras 50 épocas adicionales, en cada época se realizan 500 iteraciones, reduciendo el *learning rate* en un factor 10 para la segunda fase.

### 4.3.3 Resultados de Mask R-CNN

Los resultados obtenidos con la implementación de Mask R-CNN se presentan a continuación. Se debe tener en cuenta que el entrenamiento se parte como se ha mencionado en dos fases, pero las gráficas que se van a mostrar incorporan ambas fases. En este caso dado el largo tiempo de ejecución del modelo se ha decidido solo imprimir las gráficas más importantes que son las de la función de pérdida y la del mIoU.

A continuación, se muestra una gráfica con la convergencia de la función de pérdida global de la red a lo largo de las 100 épocas:

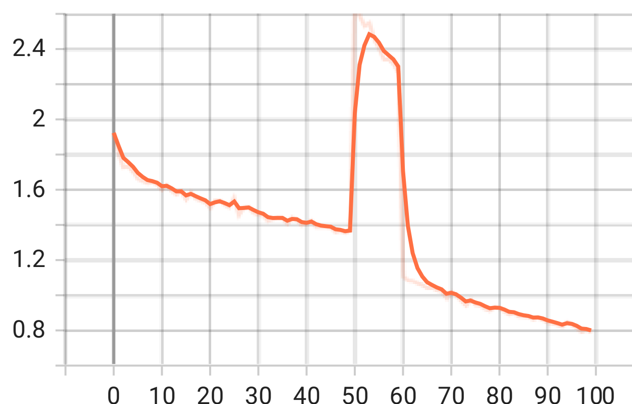



*Figura 63. Gráfica de la evolución de la función de pérdida de Mask R-CNN*

Como se observa en la gráfica la red va reduciendo su error de manera casi lineal hasta la época 50, momento en el cual se pasa a la segunda fase en la que se observa un aumento del error, en este caso es completamente normal, ya que en la primera fase solo se estaban entrenando las cabezas de la red por

lo que al introducir todo el resto de los parámetros de la red en la segunda fase se espera un incremento del error al tener efecto sobre la función de pérdida. De todas formas se observa también como tras unas 10 épocas desde que empieza la segunda fase el error va disminuyendo a valores más bajos que la primera fase lo cual indica un mejor aprendizaje en ambas fases.

El mIoU obtenido con este modelo se muestra en la siguiente gráfica:

Figura 64. Gráfica de la evolución del mIoU de Mask R-CNN

Podemos ver que este es el modelo que más mIoU obtiene, rondando valores próximos a 0.64. Se aprecia mayores valores al principio (alrededor de 0.67), probablemente porque este modelo converge muy rápido sobre el *dataset* y va estabilizándose a valores más bajos seguramente por un sobreajuste a los datos. Hay que tener en cuenta que en este caso no se suman a la media las clases más comunes que podrían subirla un poco como lo son la clase acera, carretera o árboles, ya que en este caso hay menos clases de instancia y que hacen alusión a elementos más específicos.

Los resultados proporcionados por el modelo en distintas etapas cogiendo imágenes del *dataset* de validación se muestran a continuación:

- En la época 1 la función objetivo toma el valor de 1.924 obteniéndose las imágenes mostradas en la figura 65:

Figura 65. Resultados de Mask R-CNN en la época 1

- En la época 50 la función objetivo toma el valor de 1.371 obteniéndose las imágenes mostradas en la figura 66:

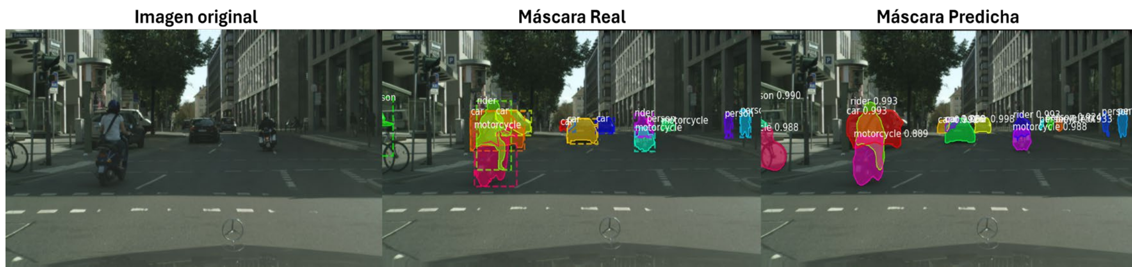



*Figura 66. Resultados de Mask R-CNN en la época 50*

- En la época 100 la función objetivo toma el valor de 0.1381 obteniéndose las imágenes mostradas en la figura 67:

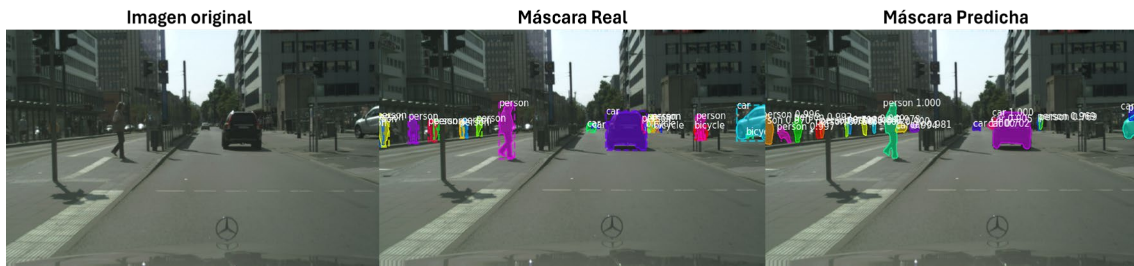



*Figura 67. Resultados de Mask R-CNN en la época 100*

En este caso se comprueba como la red Mask R-CNN es la que mejores resultados produce en términos de delineado y de reconocimiento de objetos lejanos (aunque se debe diferencia lo que es segmentación semántica de instancia) y lo hace desde épocas tempranas, siendo capaz de distinguir muy bien las clases de instancia con las que es entrenada.

Tal es el caso que hay imágenes en las que el modelo obtiene de resultados máscaras para instancias que en el *dataset* de validación no se especifican y que sí que son correctas como por ejemplo en la siguiente imagen obtenida en la época 29:

*Figura 68. Resultados de Mask R-CNN en la época 29*

Como se aprecia en la imagen de la izquierda según las anotaciones del *dataset* de validación ahí no debería de haber ningún coche (aunque sí que lo hay) mientras que en la imagen de la derecha el modelo predice correctamente que sí que hay un coche.

## 4.4 Comentario de resultados

Este apartado plasma el resumen de los resultados obtenidos con los distintos modelos.

El mIoU máximo reportado por parte de cada modelo se muestra en la siguiente figura:

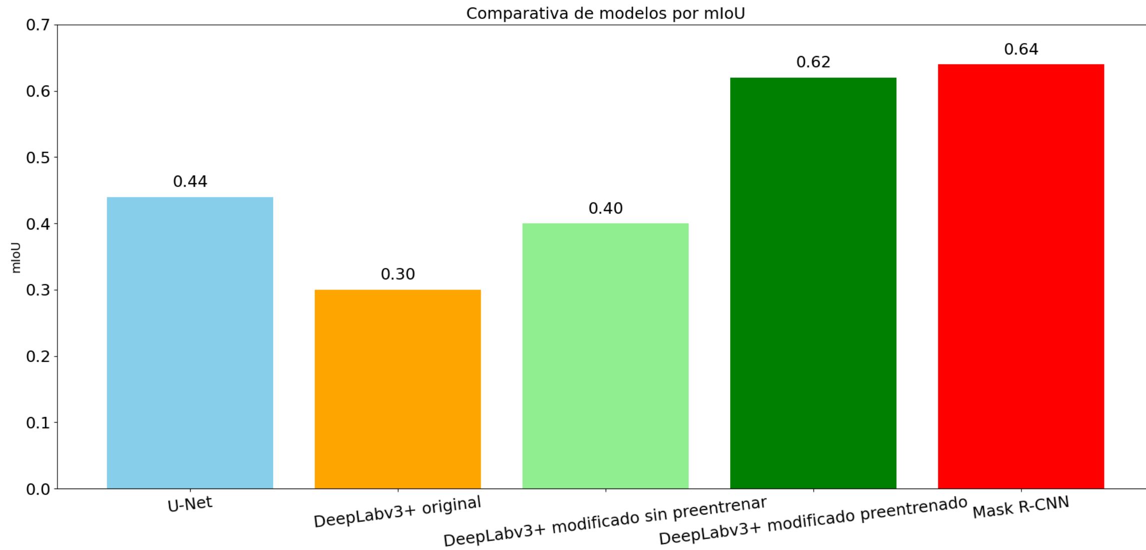



*Figura 69. Resumen del mIoU máximo de cada modelo*

Tal y como se observa en la figura, los modelos que alcanzan un mayor valor de mIoU son DeepLabv3+ modificado con preentrenamiento y Mask R-CNN, ambos con puntuaciones en torno a 0.6. En contraste, el modelo DeepLabv3+ original sin preentrenamiento muestra el rendimiento más bajo, con un valor de mIoU sensiblemente inferior, aproximadamente un 50 % menor respecto al mejor modelo, lo que pone de manifiesto las limitaciones de esta arquitectura cuando no se dispone de pesos preentrenados, igual que pasa con DeepLabv3+ modificado sin preentrenamiento, aunque en este caso la diferencia no es tan drástica siendo su mIoU de 0.4.

Por su parte, el modelo U-Net se sitúa en un término intermedio entre los mejores y el peor resultado, logrando un desempeño razonable dadas sus características de simplicidad arquitectónica y bajo coste computacional.

Un resumen de las máscaras de segmentación producidas por los modelos que contrasta con lo anterior se ilustra a continuación:

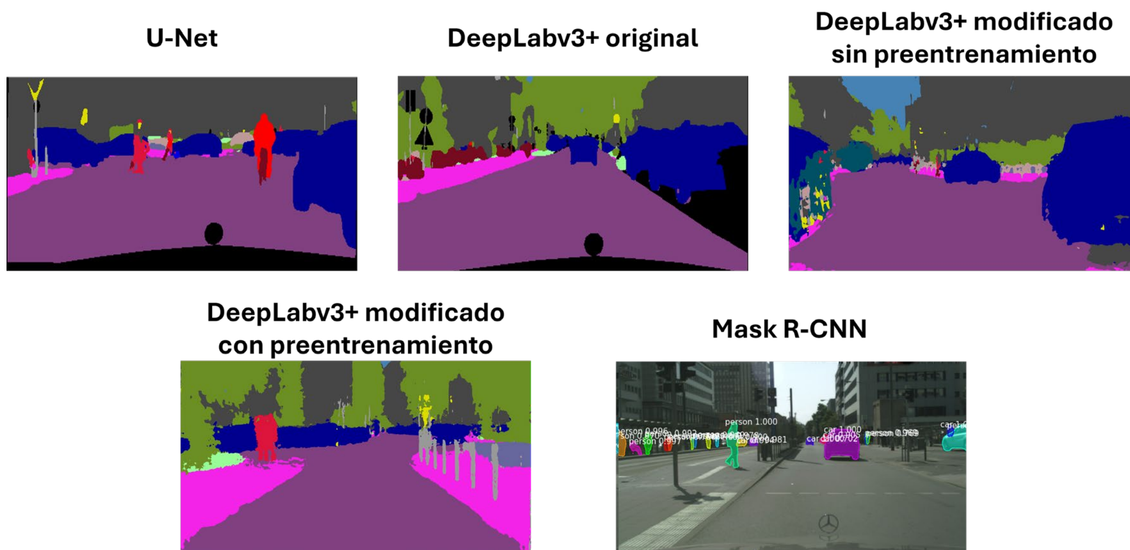



*Figura 70. Máscaras devueltas por los distintos modelos*

Como se ve el resultado es el esperado con respecto a los valores de mIoU obtenidos.

En cuanto a los tiempos de ejecución obtenidos, a continuación se presentan los resultados que reflejan el consumo de recursos en términos de tiempo de cómputo asociado a cada modelo (se debe tener en cuenta que todos los modelos se han ejecutado con la A100 de Nvidia, excepto el modelo U-Net que para su ejecución se ha empleado una GTX 1660 TI):

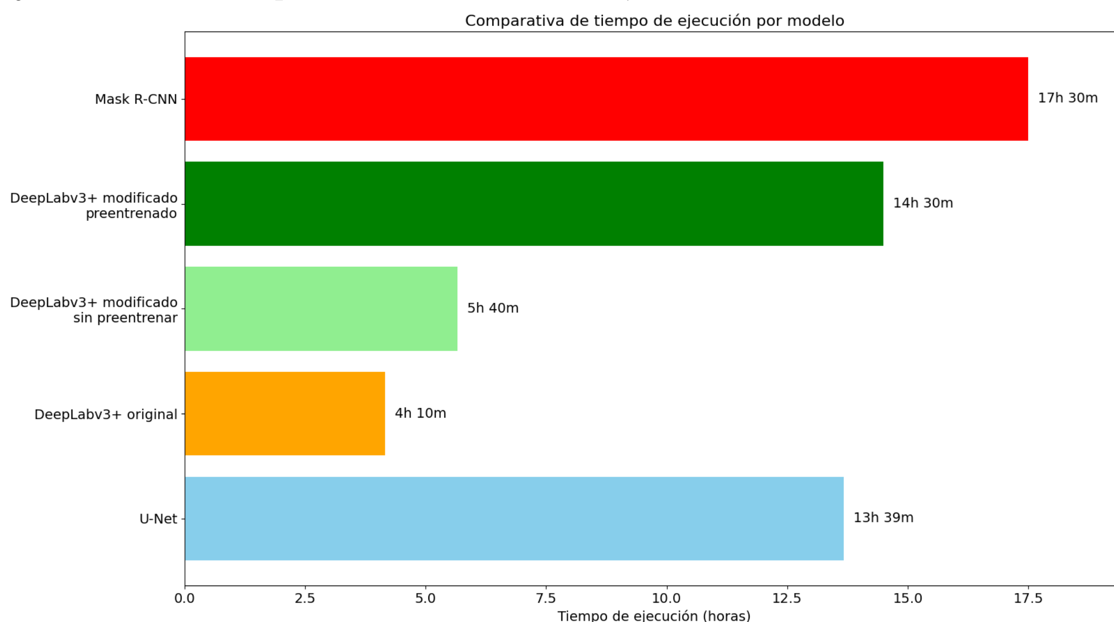



*Figura 71. Tiempo de ejecución de los distintos modelos*

Los resultados también se ajustan a los valores de mIoU alcanzados por cada modelo. No obstante, si se considera conjuntamente el rendimiento (mIoU) y el tiempo de ejecución, U-Net se posiciona como la opción más eficiente, especialmente teniendo en cuenta que fue ejecutado en una GPU menos potente (GTX 1660 TI).

# 5. Conclusiones

A lo largo de este trabajo de fin de máster se han implementado y evaluado tres modelos de segmentación basados en arquitecturas de creciente complejidad: U-Net, DeepLabv3+ (implementación del artículo original sin preentrenamiento), DeepLabv3+ (versión de un repositorio de GitHub sin y con preentrenamiento) y Mask R-CNN. Todas las redes se han desarrollado en PyTorch, excepto Mask R-CNN que utiliza un código desarrollado en TensorFlow 2.14. Los modelos se han entrenado empleando las GPUs GTX 1660 TI y la A100 de Nvidia. Cada uno se ha ajustado en base a prueba error escogiendo cuidadosamente los hiperparámetros utilizados, desde el tamaño de *batch* hasta la política de *learning rate* con el objetivo de maximizar su rendimiento en mIoU sobre el *dataset* de validación.

Es difícil concluir que modelo consigue mejores resultados ya que cada uno presenta unas peculiaridades distintas.

La red U-Net ha demostrado ser la opción más ligera y rápida, pudiendo ser entrenada en pocas horas con una GPU menos potente que la utilizada en el resto de los modelos. Su sencillo esquema de codificador-decodificador con conexiones de salto le permite converger en pocos minutos y con recursos modestos, resultando especialmente adecuada cuando se dispone de pocos datos o GPUs limitadas. Aunque su capacidad de captar contexto global es menor siendo su mIoU de entorno al 0.44, con estos datos, su facilidad de implementación y velocidad de entrenamiento la convierten en una excelente base de comparación y un punto de partida robusto para tareas de segmentación semántica sencillas.

El modelo DeepLabv3+ sin preetrenamiento puso de manifiesto la complejidad inherente a su *backbone* Xception modificado. Sin pesos ajustados, requiere entrenamiento prolongado para generalizar, cosa que no llega a hacer del todo bien con un *dataset* pequeño como lo es Cityscapes, lo que a su vez implica un mayor consumo de memoria y tiempo de cómputo, alcancanzándose unos resultados mediocres, de tan solo 0.3 de mIoU (tan solo se consigue sobre el promedio de las clases acertar el 30 % de píxeles), para dicho consumo de recursos. Se espera que con pesos preentrenados el modelo mejore significativamente como indican en el artículo original. En cambio, la versión de GitHub mostró que un *backbone* más ligero y un preprocesado con *augmentations* acelera la convergencia y mejora la calidad de las máscaras, sobre todo si esto va acompañado por un *backbone* preentrenado con un *dataset* grande como el de ImageNet (parte que marca la diferencia), DeepLabv3+ exhibe un gran salto en robustez y precisión como se observa en las gráficas de mIoU ya que se llega a alcanzar un valor de 0.62 con un correcto preentrenamiento, aunque requiere de muchos recursos para su entrenamiento.

Por su parte, Mask R-CNN brinda la flexibilidad de la segmentación de instancia. A pesar de su pipeline más complejo, ofrece resultados de muy alta fidelidad en la delineación de objetos individuales y la predicción simultánea de *bounding boxes*. Su coste computacional elevado lo hace ideal cuando se requiere de gran precisión en la segmentación, especialmente cuando se busca segmentación de instancia. Este es el modelo que muestra la mayor tasa de aciertos en la predicción siendo su mIoU de 0.64.

Cada modelo aporta beneficios claros según el escenario de uso: U-Net para implementaciones rápidas y con limitados recursos; DeepLabv3+ para un mejor equilibrio entre detalle local y contexto global, especialmente cuando se cuenta

con preetrenamiento; y Mask R-CNN cuando se precisa extraer instancias concretas con máscaras precisas a costa de mayor coste computacional.

# 6. Líneas futuras

Este trabajo de fin de máster sienta las bases para múltiples extensiones y mejoras en el ámbito de la segmentación de imágenes. Entre las posibles líneas de investigación y desarrollo se incluyen:

- **Arquitecturas ligeras y cuantización para *edge computing*:** Optimizar modelos como U-Net o DeepLabv3+ para dispositivos con recursos limitados (drones, cámaras inteligentes, sistemas embebidos) es fundamental. El objetivo es conseguir segmentación precisa en tiempo real sin depender de la nube, habilitando aplicaciones de vigilancia autónoma, inspección aérea y análisis en campo.
- **Integración de datos multimodales:** La fusión de imágenes RGB con nubes de puntos LIDAR o mapas de profundidad puede aportar una comprensión espacial más rica y mejorar la robustez de la segmentación en escenas 3D complejas. Se pueden diseñar codificadores paralelos o módulos de atención cruzada sobre los modelos presentados que procesen ambas modalidades de forma conjunta, con aplicaciones directas en vehículos autónomos, robótica móvil e inspección industrial.
- **Agricultura de precisión y monitoreo ambiental:** Aplicar estos modelos sobre imágenes satelitales o capturadas por drones para segmentar cultivos, identificar plagas y evaluar la salud de las plantas abre un campo de gran impacto. Se pueden entrenar redes especializadas en distinguir diferentes tipos de vegetación, suelo expuesto o síntomas de enfermedades, y combinarlo con análisis temporal para detectar patrones de crecimiento o estrés hídrico. Además, incorporar IA en drones permite la detección temprana de incendios forestales y la vigilancia de ecosistemas, mejorando la eficiencia en la gestión de recursos agrícolas y la prevención de desastres.

# Bibliografía


[1] S. A. Alowais et al., “Revolutionizing healthcare: The role of artificial intelligence in clinical practice,” BMC Medical Education, vol. 23, no. 1, Sep. 2023.

[2] A. H. Salem, S. M. Azzam, O. E. Emam, and A. A. Abohany, “Advancing cybersecurity: a comprehensive review of AI-driven detection techniques,” Journal Of Big Data, vol. 11, no. 1, pp. 1–38, Aug. 2024.

[3] S. Ghazal, A. Munir, and W. S. Qureshi, “Computer vision in smart agriculture and precision farming: Techniques and applications,” Artificial Intelligence in Agriculture, vol. 13, pp. 64-83, Jun. 2024.

[4] Dong, Xingshuai, and Massimiliano Cappuccio. Applications of Computer Vision in Autonomous Vehicles: Methods, Challenges and Future *Directions*. 15 Nov. 2023.

[5] R. Velastegui, Maxim Tatarchenko, Sezer Karaoglu, and T. Gevers, “Image semantic segmentation of indoor scenes: A survey,” Computer Vision and Image Understanding, vol. 248, pp. 104102–104102, Jul. 2024.

[6] C. Cărunta, A. Cărunta, and C.-A. Popa, “Heavy and Lightweight Deep Learning Models for Semantic Segmentation: A Survey,” IEEE Access, vol. 13, pp. 17745–17765, 2025.

[7] J. Long, E. Shelhamer, and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” arXiv.org, *arXiv:1411.4038*, 2014.

[8] V. Badrinarayanan, A. Kendall, and R. Cipolla, “SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation,” arXiv.org, arXiv:1511.00561, Oct. 2016.

[9] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” arXiv.org, arXiv:1505.04597, May 18, 2015.

[10] Z. Zhou, M. Rahman, N. Tajbakhsh, and J. Liang, “UNet++: A Nested U-Net Architecture for Medical Image Segmentation,” arXiv.org, arXiv: 1807.10165, 2018.

[11] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, “Pyramid Scene Parsing Network,” arXiv.org, arXiv:1612.01105, Apr. 27, 2017.

[12] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, “Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs,” arXiv.org, arXiv:1412.7062, 2014.

[13] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, “DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs,” arXiv.org (Cornell University), arXiv:1606.00915, Jun. 2016.

[14] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, “Rethinking Atrous Convolution for Semantic Image Segmentation,” arXiv.org, arXiv:1706.05587, Jun. 2017.

[15] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, “Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation,” arXiv.org, arXiv:1802.02611, 2018.

[16] J. Fu et al., “Dual Attention Network for Scene Segmentation,” arXiv.org, arXiv:1809.02983, Sep. 2018.

[17] S. Zheng et al., “Rethinking Semantic Segmentation from a Sequence-to-Sequence Perspective with Transformers,” arXiv.org, arXiv:2012.15840, Dec. 2021.

[18] E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez, and P. Luo, “SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers,” arXiv.org, arXiv:2105.15203 , Oct. 2021.

[19] R. Strudel, R. Garcia, I. Laptev, and C. Schmid, “Segmenter: Transformer for Semantic Segmentation,” arXiv.org, arXiv:2105.05633, Sep. 2021.

[20] D. Zhou et al., “Understanding The Robustness in Vision Transformers,” arXiv.org, arXiv:2204.12451, 2022.

[21] Y. Guo, D. Stutz, and B. Schiele, “Robustifying Token Attention for Vision Transformers,” arXiv.org, arXiv:2303.11126, 2023.

[22] W. Wang et al., “InternImage: Exploring Large-Scale Vision Foundation Models with Deformable Convolutions,” arXiv.org, arXiv:2211.05778, Apr. 2023.

[23] S. Erisen, “SERNet-Former: Semantic Segmentation by Efficient Residual Network with Attention-Boosting Gates and Attention-Fusion Networks,” arXiv.org, arXiv:2401.15741, 2024.

[24] R. Sharma, M. Saqib, C. T. Lin, and M. Blumenstein, “A Survey on Object Instance Segmentation,” SN Computer Science, vol. 3, n° art 499, Sep. 2022.

[25] Soumyadip, Shubham, and B. Sharma, “Mastering All YOLO Models from YOLOv1 to YOLO-NAS: Papers Explained (2024),” learnopencv.com, Dec. 26, 2023. https://learnopencv.com/mastering-all-yolo-models/

[26] J. Dai, K. He, Y. Li, S. Ren, and J. Sun, “Instance-sensitive Fully Convolutional Networks,” arXiv.org, arXiv:1603.08678, 2016.

[27] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask R-CNN,” arXiv.org, arXiv:1703.06870, 2017.

[28] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, “Path Aggregation Network for Instance Segmentation,” arXiv.org, arXiv:1803.01534, Sep. 2018.

[29] K. Chen et al., “Hybrid Task Cascade for Instance Segmentation,” arXiv.org, arXiv:1901.07518, Apr. 2019.

[30] Bolya D, Zhou C, Xiao F, Lee Y. Yolact: real-time instance segmentation. In: Proceedings of the IEEE/CVF international conference on computer vision, pp. 9157–9166. 2019.

[31] M. Bai and R. Urtasun, “Deep Watershed Transform for Instance Segmentation,” arXiv.org, arXiv:1611.08303, Nov. 24, 2016.

[32] C.-Y. Fu, M. Shvets, and A. C. Berg, “RetinaMask: Learning to predict masks improves state-of-the-art single-shot detection for free,” arXiv.org, arXiv:1901.03353, Jan. 2019.

[33] E. Xie et al., “PolarMask: Single Shot Instance Segmentation with Polar Representation,” arXiv.org, arXiv:1909.13226, 2019.

[34] Zhi Tian, Chunhua Shen, Hao Chen, and Tong He. FCOS: Fully convolutional one-stage object detection. In Proc. IEEE Int. Conf. Comp. Vis., pp. 9626-9635, 2019.

[35] X. Wang, T. Kong, C. Shen, Y. Jiang, and L. Li, “SOLO: Segmenting Objects by Locations,” arXiv (Cornell University), arXiv:1912.04488, Dec. 2019.

[36] X. Wang, R. Zhang, T. Kong, L. Li, and C. Shen, “SOLOv2: Dynamic and Fast Instance Segmentation,” arXiv.org, arXiv:2003.10152, Oct. 23, 2020.

[37] J. Hu et al., “ISTR: End-to-End Instance Segmentation with Transformers,” arXiv.org, arXiv:2105.00637, 2021.

[38] R. Guo, D. Niu, L. Qu, and Z. Li, “SOTR: Segmenting Objects with Transformers,” arXiv.org, arXiv:2108.06747, 2021.

[39] X. Yu, D. Shi, X. Wei, Y. Ren, T. Ye, and W. Tan, “SOIT: Segmenting Objects with Instance-Aware Transformers,” arXiv.org, arXiv:2112.11037, 2021.

[40] B. Cheng, I. Misra, A. G. Schwing, A. Kirillov, and R. Girdhar, “Masked-attention Mask Transformer for Universal Image Segmentation,” arXiv.org, arXiv:2112.01527, Dec. 2021.

[41] Y. Fang et al., “Instances as Queries,” arXiv.org, arXiv:2105.01928, 2021.

[42] T. Cheng et al., “Sparse Instance Activation for Real-Time Instance Segmentation,” arXiv.org, arXiv:2203.12827, 2022.

[43] A. Kirillov et al., “Segment Anything,” arXiv.org, arXiv:2304.02643, Apr. 2023.

[44] D. Reis, J. Kupec, J. Hong, and A. Daoudi, “Real-Time Flying Object Detection with YOLOv8,” arXiv.org, arXiv:2305.09972, May 2023.

[45] C.-Y. Wang, I-Hau. Yeh, and H.-Y. M. Liao, “YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information,” arXiv (Cornell University), arXiv:2402.13616, Feb. 2024.

[46] A. Wang et al., “YOLOv10: Real-Time End-to-End Object Detection,” arXiv.org, arXiv:2405.14458, May 23, 2024.

[47] T. Ren et al., “Grounded SAM: Assembling Open-World Models for Diverse Visual Tasks,” arXiv.org, arXiv:2401.14159, 2024.

[48] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,” in Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations, D. E. Rumelhart and J. L. McClelland, Eds. Cambridge, MA, USA: MIT Press, 1986, pp. 318–362.

[49] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” arXiv.org, arXiv:1409.1556, Apr. 10, 2015.

[50] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” arXiv.org, arXiv:1512.03385, Dec. 10, 2015.

[51] F. Chollet, “Xception: Deep Learning with Depthwise Separable Convolutions,” arXiv.org, arXiv:1610.02357, 2016.

[52] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," arXiv.org, arXiv:1311.2524, 2013.

[53] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders, "Selective Search for Object Recognition," International Journal of Computer Vision, vol. 104, no. 2, pp. 154–171, Apr. 2013.

[54] P. F. Felzenszwalb and D. P. Huttenlocher, "Efficient Graph-Based Image Segmentation," International Journal of Computer Vision, vol. 59, no. 2, pp. 167–181, Sep. 2004.

[55] W. T. Freeman and E. H. Adelson, "The design and use of steerable filters," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 13, no. 9, pp. 891–906, 1991.

[56] Roberto Valle, "Distribución Gaussiana," Gist, Dec. 14, 2024. https://gist.github.com/bobetocalo/e302fed4ca19c9b7807a

[57] J. Maucher, "Gaussian Filter and Derivatives of Gaussian — Object Recognition Lecture," Hdm-stuttgart.de, 2024. https://maucher.pages.mi.hdm-stuttgart.de/orbook/preprocessing/04gaussianDerivatives.html

[58] A. Torralba, "Lecture 3 Linear filters," 2016. Available: http://6.869.csail.mit.edu/fa16/lecture/lecture3linearfilters.pdf

[59] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," Communications of the ACM, vol. 60, no. 6, pp. 84–90, May 2012.

[60] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," 2009 IEEE Conference on Computer Vision and Pattern Recognition, pp. 248-255, Jun. 2009.

[61] R. Girshick, "Fast R-CNN," arXiv.org, arXiv:1504.08083, 2015.

[62] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," arXiv.org, arXiv:1506.01497, Jun. 04, 2015.

[63] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature Pyramid Networks for Object Detection," arXiv:1612.03144, Apr. 2017.

[64] "Login – Cityscapes Dataset," Cityscapes-dataset.com, 2020. https://www.cityscapes-dataset.com/downloads

[65] G. Fang, "DeepLabv3Plus-Pytorch," GitHub, Jun. 23, 2023. https://github.com/VainF/DeepLabV3Plus-Pytorch

[66] Matterport, "matterport/Mask_RCNN," GitHub, Mar. 29, 2020. https://github.com/matterport/Mask_RCNN

[67] z-mahmud22, "GitHub - z-mahmud22/Mask-RCNN_TF2.14.0: Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow 2.14.0 and Python 3.10.," GitHub, 2023. https://github.com/z-mahmud22/Mask-RCNN_TF2.14.0

[68] K. Erdem, "Understanding Region of Interest (RoI Pooling) - Blog by Kemal Erdem," https://erdem.pl, Feb. 2020. https://erdem.pl/2020/02/understanding-region-of-interest-ro-i-pooling

# Apéndice

A lo largo de este capítulo se explican con mayor detalle algunos conceptos con la finalidad de hacerlos más comprensibles.

## A.1

Un ejemplo del funcionamiento de CRF se puede establecer partiendo de la suposición de que se tiene una imagen de 3x3 píxeles. La red produce, para cada píxel, la probabilidad de que pertenezca a la clase "objeto" (la otra, fondo, es 1 menos esa probabilidad). Por ejemplo, para la clase "objeto" se podría tener la siguiente matriz de probabilidades:

$$P_{objeto} = \begin{bmatrix} 0.9 & 0.8 & 0.2 \\ 0.7 & 0.6 & 0.3 \\ 0.1 & 0.2 & 0.4 \end{bmatrix}$$

Para un píxel concreto, por ejemplo, el de la esquina superior izquierda (posición (1,1)) la probabilidad es 0.9. La función unaria asignándolo como "objeto" será:

$$\theta_i(x_i) = -\log P(\,0.9) \approx 0.105,$$

En cambio, asignándola como fondo la probabilidad cambia a 1-0.9=0.1 y su función unaria resulta en:

$$\theta_i(x_i) = -\log P(\,0.1) \approx 2.302,$$

lo que significa que basándose únicamente en la salida de la red es más "eficiente" (en términos de energía) asignar la etiqueta "objeto" a (1,1).

Además de considerar el valor unario se debe considerar las relaciones entre píxeles. Suponiendo que se utilizan los siguientes valores:

$w_1$=1, $w_2$=1

$\sigma_\alpha$=3 (para la distancia espacial)

$\sigma_\beta$=10 (para la diferencia de color)

$\sigma_\gamma$=5,

Viendo (para este ejemplo) solo la relación entre el píxel (1,1) y su vecino (1,2), suponiendo que sus intensidades son 100 y 105 respectivamente, $\theta_{ij}(x_i, x_j)$ se calcula de la siguiente manera:

$$k_1 + k_2 = \; 1 * \exp\left(-\frac{\|(1-1)^2-(1-2)^2\|^2}{2*3^2} - \frac{\|100-105\|^2}{2*10^2}\right) +$$

$$+\,1\,\exp\left(-\frac{\|(1-1)^2-(1-2)^2\|^2}{2*5^2}\right) \approx 1.325$$

y

$$\theta_{(1,1),(1,2)}\left(x_{(1,1)}, x_{(1,2)}\right) = \mu\left(x_{(1,1)}, x_{(1,2)}\right) * (k_1 + k_2)$$

Para el caso donde (1,1) y (1,2) reciben etiquetas diferentes ((1,1) es "objeto" y (1,2) es "fondo"), se da que $\mu = 1$ lo que agrega un coste de 1.325 al total de la energía. Ahora se tiene que ver qué caso minimiza la energía, dejar (1,1) como "objeto" o como "fondo", para el caso donde (1,2) se ha etiquetado como "fondo":

Píxel (1,1) clasificado como "objeto":

$$E(x) = \sum_i \theta_i(x_i) + \sum_{ij} \theta_{ij}(x_i, x_j) = 0.105 + 1.325 = 1.43$$

Píxel (1,1) clasificado como “fondo”:

$$E(x) = \sum_i \theta_i(x_i) + \sum_{ij} \theta_{ij}(x_i, x_j) = 2.302 + 0 = 2.302 \; (ya\ que\ \mu\ es\ 0\ ahora),$$

Esto indica que para el caso donde (1,2) es “fondo” lo mejor es dejar (1,1) como “objeto”. En realidad se deben comparar todos los píxeles de la matriz con (1,1) para calcular su energía total, pero por simplicidad solo se ha hecho con (1,2) la comparación. De esta forma analizando todas las combinaciones se busca que el sumatorio de las energías de todos los píxeles sea lo menor posible.

# A.2

Para entender bien esta técnica se procede a la ejemplificación de la misma tomando una imagen en blanco y negro (1 canal), utilizando $N_4$ *neighborhood* y K=100, para mayor simplicidad:

**- Paso 1:** En este paso se muestra a través de la figura x el estado inicial de la imagen:

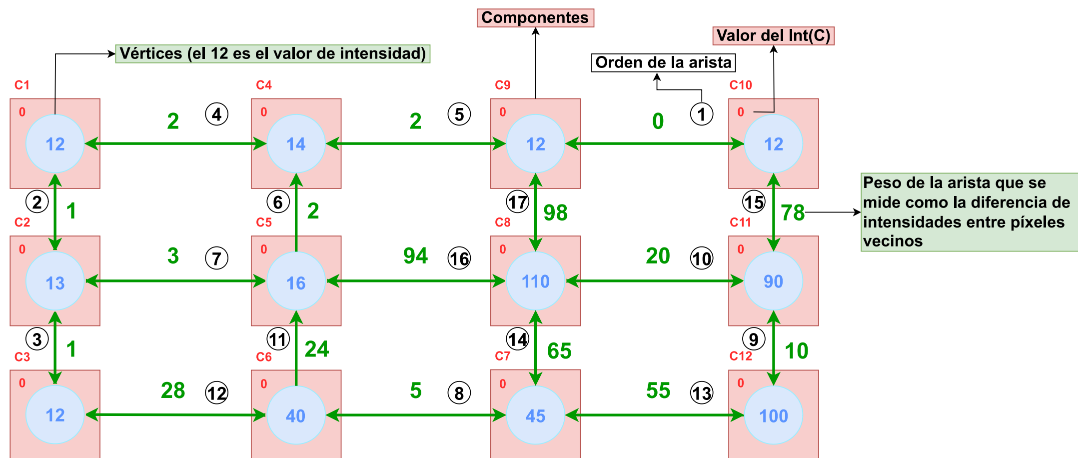



*Figura 10. Esquema del estado inicial del algoritmo de Felzenszwalb*

En este paso las aristas se ordenan por orden creciente (de menor a mayor diferencia entre intensidades de sus vértices), de esta forma el conjunto E queda E=[$e_1$, $e_2$, $e_3$,…,$e_4$].

**- Paso 2:** se procede a la unión de componentes. Para ello se toma la arista 1 ($e_1$) y se comprueba si los dos vértices/componentes que la conforman pueden ser unificados.

$Dif(C_{10}, C_9) = 0$, la única arista que conecta ambos vale 0,

$Int(C_9) = 0;\ Int(C_{10}) = 0;\ \tau(C_9) = \frac{K}{|C_9|} = \frac{100}{1} = 100;\ \tau(C_{10}) = \frac{K}{|C_{10}|} = \frac{100}{1} = 100,$

$MInt(C_9, C_{10}) = mín\big(Int(C_9) + \tau(C_9), Int(C_{10}) + \tau(C_{10})\big) = mín(0 + 100, 0 + 100) = 100,$

$$D(C_{10}, C_9) = \begin{cases} True, & si\ Dif(C_{10}, C_9) > MInt(C_{10}, C_9) \rightarrow 0 \ngtr 100 \\ & False,\ en\ caso\ contrario \end{cases}.$$

Como se observa D es falso lo que significa que no hay límite entre $C_9$ y $C_{10}$ y que, por tanto, se pueden unificar. A continuación, se muestran cómo queda tras este paso el esquema:

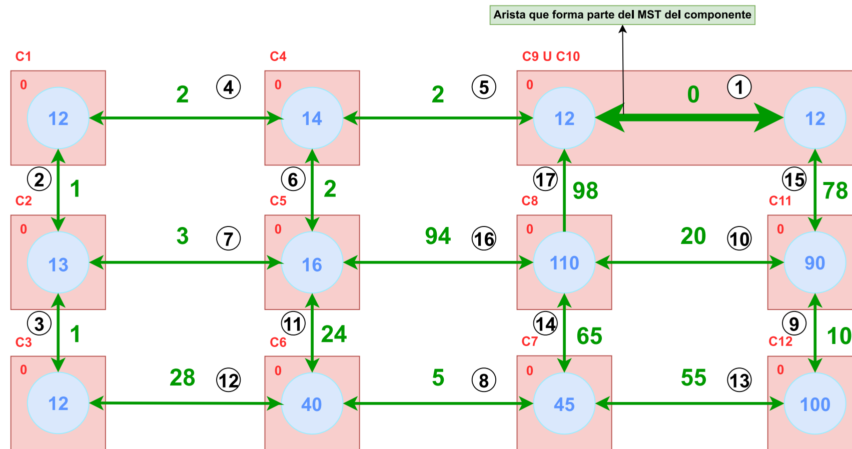



*Figura 11. Esquema del paso 2 del algoritmo de Felzenszwalb*

- **Paso 3:** A partir de este paso se realiza el mismo procedimiento que en el paso anterior, pero con la siguiente arista, en este caso la 2 ($e_2$).

$Dif(C_1, C_2) = 1$, la única arista que conecta ambos vale 1,

$Int(C_1) = 0;\ Int(C_2) = 0;\ \tau(C_1) = \frac{K}{|C_1|} = \frac{100}{1} = 100;\ \tau(C_{10}) = \frac{K}{|C_2|} = \frac{100}{1} = 100,$

$MInt(C_1, C_1) = mín\big(Int(C_1) + \tau(C_1), Int(C_2) + \tau(C_2)\big) = mín(0 + 100, 0 + 100) = 100,$

$$D(C_1, C_2) = \begin{cases} True, & si\ Dif(C_1, C_2) > MInt(C_1, C_2) \rightarrow 1 \ngtr 100 \\ False, & en\ caso\ contrario \end{cases}.$$

Como D es falso $C_1$ y $C_2$ se unifican. A continuación se muestra como resulta esta unificación:

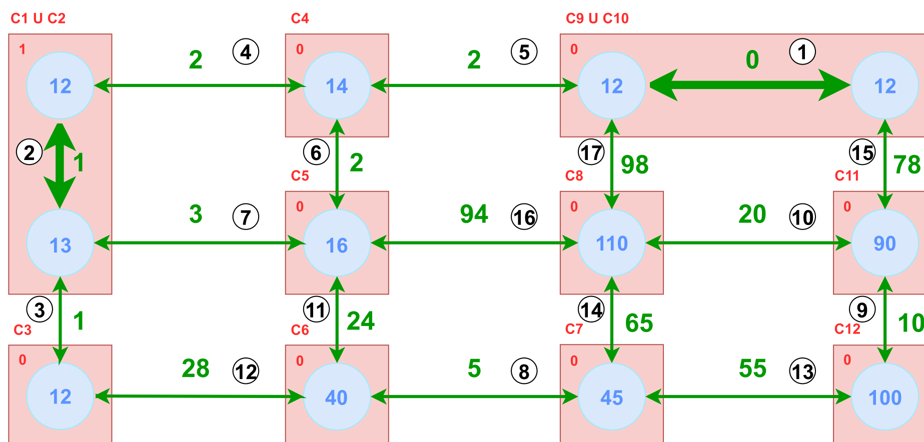



*Figura 12. Esquema del paso 3 del algoritmo de Felzenszwalb*

Tras realizar varias iteraciones del algoritmo hasta haber comprobado todas las aristas los componentes quedan de la siguiente manera:

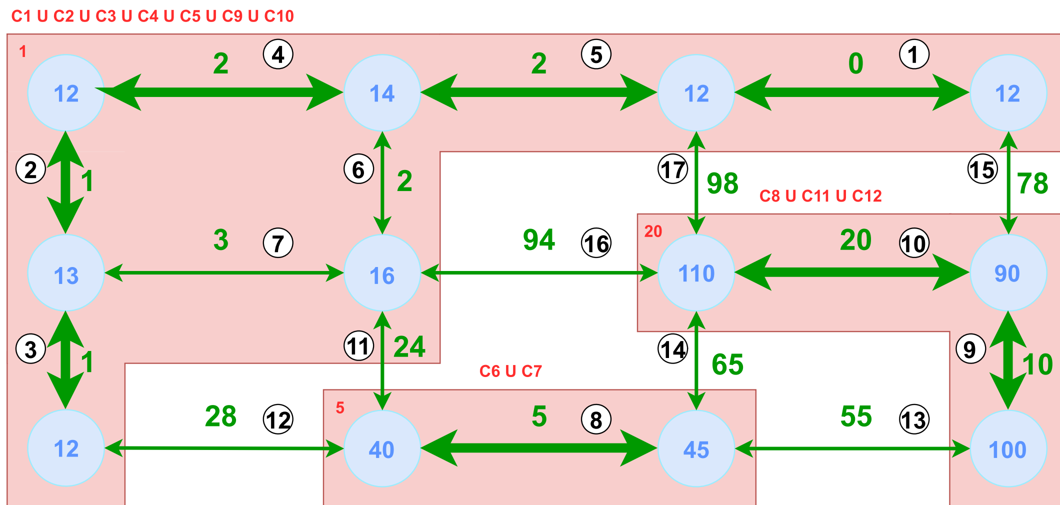
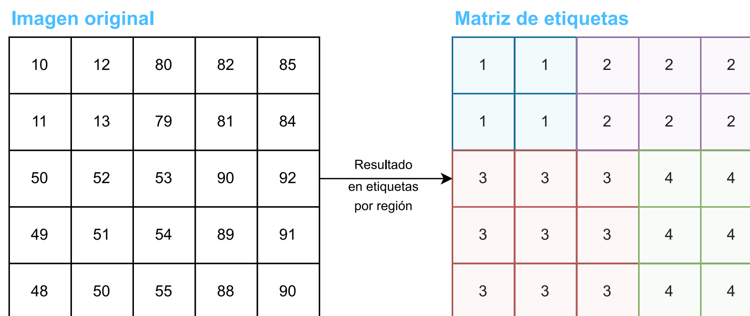



*Figura 13. Esquema del resultado final del algoritmo de Felzenszwalb*

Estos componentes finales son los que conforman las regiones de la segmentación inicial de la imagen de entrada. Con ellos el algoritmo de Felzenszwalb devuelve una matriz de etiquetas del ancho y alto de la imagen original indicando a que componente pertenece cada píxel, como se muestra en la siguiente figura:




| 10 | 12 | 80 | 82 | 85 |
|---|---|---|---|---|
| 11 | 13 | 79 | 81 | 84 |
| 50 | 52 | 53 | 90 | 92 |
| 49 | 51 | 54 | 89 | 91 |
| 48 | 50 | 55 | 88 | 90 |

| 1 | 1 | 2 | 2 | 2 |
|---|---|---|---|---|
| 1 | 1 | 2 | 2 | 2 |
| 3 | 3 | 3 | 4 | 4 |
| 3 | 3 | 3 | 4 | 4 |
| 3 | 3 | 3 | 4 | 4 |

*Figura 14. Visualización de la matriz de etiquetas*

## A.3

Tomando como ejemplo la siguiente "imagen":

| (-1,-1)=70 | (0,-1)=30 | (1,-1)=30 |
|---|---|---|
| (-1,0)=0 | (0,0)=50 | (1,0)=60 |
| (-1,1)=20 | (0,1)=30 | (1,1)=90 |

A continuación se explica como calcular las derivadas gaussianas sobre el píxel central (0,0)=50 en una orientación:

Dada la función Gaussiana bidimensional para σ=1 (lo que usan en el artículo):

$G(x,y) = \frac{1}{2\pi}\exp\left(-\frac{x^2+y^2}{2}\right)$, cuyas derivadas parciales son:

- Respecto de x: $G_x(x,y) = \frac{\partial G}{\partial x} = -\frac{x}{2\pi}\exp\left(-\frac{x^2+y^2}{2}\right)$
- Respecto de y: $G_y(x,y) = \frac{\partial G}{\partial y} = -\frac{y}{2\pi}\exp\left(-\frac{x^2+y^2}{2}\right)$

Se obtiene la respuesta del filtro en una orientación (dirección θ) de la siguiente manera:

$G_\theta(x,y) = \cos(\theta) * G_x(x,y) + \mathrm{sen}(\theta) * G_y(x,y)$

Con estas ecuaciones ya se puede calcular el valor del filtro aplicado en una dirección. Para calcular el valor del filtro en el píxel central (0,0) para el ángulo 0, por ejemplo, se realizan los siguientes pasos:

- **Paso 1:** Cálculo de las derivadas parciales de las coordenadas de los píxeles de la matriz vecinos incluido el central:
  - En (-1,-1):

    $$G_x(-1,-1) = G_y(-1,-1) = -\frac{-1}{2\pi}\exp\left(-\frac{-1^2+-1^2}{2}\right) \approx 0.0586$$

  - En (0,-1):

    $$G_x(0,-1) = -\frac{0}{2\pi}\exp\left(-\frac{0^2+-1^2}{2}\right) = 0$$

    $$G_y(0,-1) = -\frac{-1}{2\pi}\exp\left(-\frac{0^2+-1^2}{2}\right) \approx 0.0966$$

    Y así con el resto en adelante (se omite todo el procedimiento para menos repetición).

  - En (1,-1): $G_x(1,-1) = -0.0586$; $G_y(1,-1) = 0.0586$
  - En (-1,0): $G_x(-1{,}0) = 0.0966$; $G_y(-1{,}0) = 0$
  - En (0,0): $G_x(0{,}0) = 0; G_y(0{,}0) = 0$
  - En (1,0): $G_x(1{,}0) = -0.0966$; $G_y(1{,}0) = 0$
  - En (-1,1): $G_x(-1{,}1) = 0.0586$; $G_y(-1{,}1) = -0.0586$
  - En (0,1): $G_x(0{,}1) = 0$; $G_y(0{,}1) = -0.0966$
  - En (1,1): $G_x(1{,}1) = -0.0586$; $G_y(1{,}1) = -0.0586$
- **Paso 2:** Cálculo de la respuesta del filtro en el píxel central para el ángulo 0. Primero se debe conocer la siguiente ecuación:

  $R_\theta = \sum_{(x,y)} I(x,y) * G_\theta(x,y)$, siendo I(x,y) el valor de la intensidad del píxel en (x,y), por lo que para el ejemplo planteado el filtro queda:

  $R_0 = \sum_{(x,y)} I(x,y) * G_0(x,y) = 4{,}102 + 0 - 1.758 + 0 + 0 - 5.796 + 1.172 + 0 - 5.274 = -7.55$, ese será el valor del píxel (0,0) en la orientación 0º tras aplicar el filtro.

  Esto se hace para cada píxel en las 8 orientaciones para los tres canales en la imagen RGB (en los píxeles de los bordes se rodean los vecinos inexistentes con píxeles de valor 0 de intensidad).

## A.4

A continuación se muestra un ejemplo sobre cómo hacer ROI Align [68]. En la siguiente imagen se muestra la ROI base mapeada en un mapa de características:

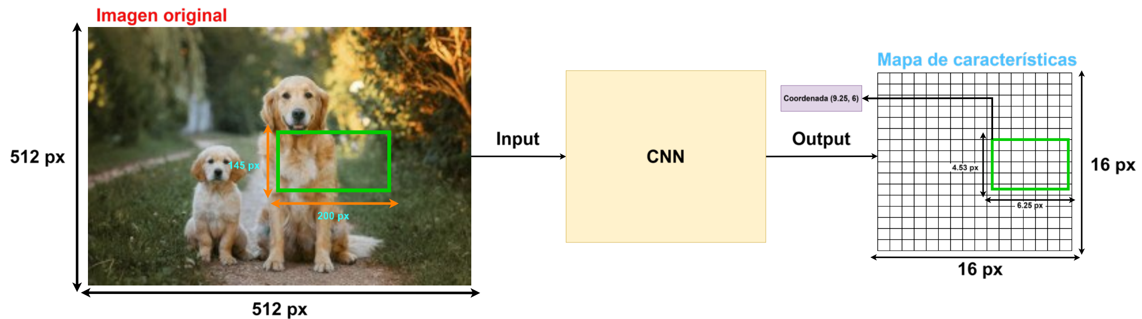



*Figura X. ROI mapeada sobre el mapa de características*

Lo que se quiere es mediante ROI *Align* transformar esa ROI rectangular a una matriz 3x3. Para ello primero se debe dividir esa ROI en 9 celdas del mismo tamaño. El cálculo del tamaño de cada celda se describe a continuación:

- Ancho de la celda:

$$width_{celda} = \frac{width_{ROI}}{3} = \frac{6.25}{3} = 2.08$$

- Alto de la celda:

$$heigth_{celda} = \frac{heigth_{ROI}}{3} = \frac{4.53}{3} = 1.51$$

Con esto ya se puede ver la ROI dividida:

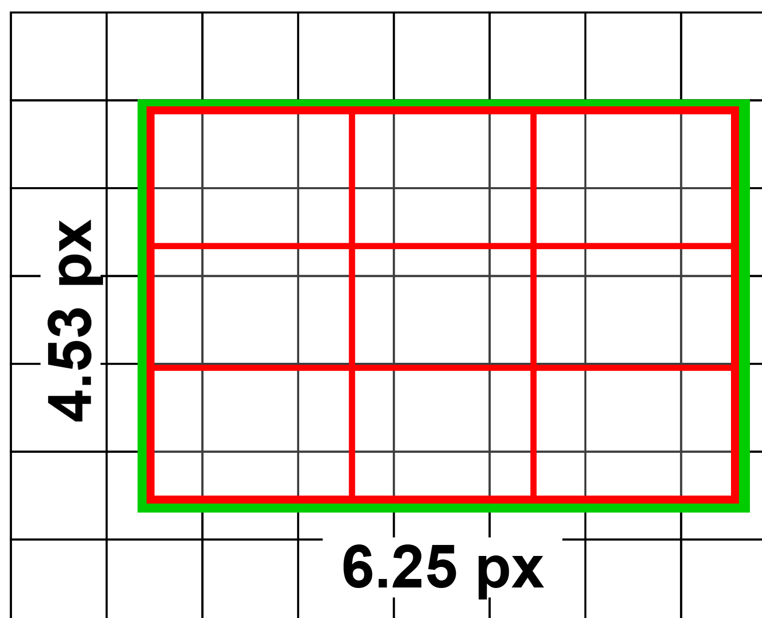



*Figura X. ROI dividida en 3x3*

Como se observa por ejemplo la celda superior izquierda cubre seis celdas diferentes por lo que para extraer su valor se deben muestrear 4 puntos en la celda. Las coordenadas de esos cuatro puntos se pueden extraer dividiendo la celda en 3. A continuación se detalla como:

- Punto 1 (arriba a la izquierda):

- Coordenada x: $x = x_{box} + \frac{width}{3} * 1 = 9.25 + \frac{2.08}{3} * 1 \approx 9.94$
- Coordenada y: $y = y_{box} + \frac{heigth}{3} * 1 = 6 + \frac{1.51}{3} * 1 \approx 6.5$

Por tanto la coordenada es (9.94, 6.5)

- Punto 2 (abajo a la izquierda):

- Coordenada x: $x = x_{box} + \frac{width}{3} * 1 = 9.25 + \frac{2.08}{3} * 1 \approx 9.94$
- Coordenada y: $y = y_{box} + \frac{heigth}{3} * 2 = 6 + \frac{1.51}{3} * 2 \approx 7.01$

Por tanto la coordenada es (9.94, 7.01)

- Punto 3 (arriba a la derecha):

- Coordenada x: $x = x_{box} + \frac{width}{3} * 2 = 9.25 + \frac{2.08}{3} * 2 \approx 10.64$
- Coordenada y: $y = y_{box} + \frac{heigth}{3} * 2 = 6 + \frac{1.51}{3} * 1 \approx 6.5$

Por tanto la coordenada es (10.64, 7.01)

- Punto 4 (abajo a la derecha):

- Coordenada x: $x = x_{box} + \frac{width}{3} * 1 = 9.25 + \frac{2.08}{3} * 2 \approx 10.64$
- Coordenada y: $y = y_{box} + \frac{heigth}{3} * 2 = 6 + \frac{1.51}{3} * 2 \approx 7.01$

Por tanto la coordenada es (10.64, 7.01)

Una vez se obtienen los cuatro puntos se puede aplicar interpolación bilineal para muestrear datos para esta celda mediante la siguiente ecuación:

$$P \approx \frac{y_2 - y}{y_2 - y_1}\left(\frac{x_2 - x}{x_2 - x_1} Q_{11} + \frac{x - x_1}{x_2 - x_1} Q_{21}\right) + \frac{y - y_1}{y_2 - y_1}\left(\frac{x_2 - x}{x_2 - x_1} Q_{12} + \frac{x - x_1}{x_2 - x_1} Q_{22}\right).$$

Con la que se debe ir punto a punto (de los cuatro obtenidos) calculando sus P. En la ecuación lo que se hace es conectar al punto escogido con el centro de las celdas vecinas más cercanas. En la ecuación y es la coordenada y del punto seleccionado, $y_2$ la coordenada y con mayor valor de entre todos los vecinos, $y_1$ lo mismo pero la de menor valor y pasa igual con $x_1$ y $x_2$. Los valores Q representan los valores que toman las casillas de esos vecinos (valor de sus píxeles). Por tanto si se quiere calcular la interpolación con el primer punto (9.94, 6.5) primero se deben indicar cuales son las coordenadas de sus vecinos más próximos (los centros de las celdas más próximas).

**INTERPOLACIÓN PUNTO 1 CELDA 1:**

- Vecinos del punto 1:

- Arriba a la izquierda: (9.5,6.5)

- Abajo a la izquierda: (9.5,7.5)
- Arriba a la derecha: (10.5,6.5)
- Abajo a la derecha: (10.5,7.5)

En la siguiente figura se ilustra al punto 1 que se va a interpolar junto a sus vecinos:

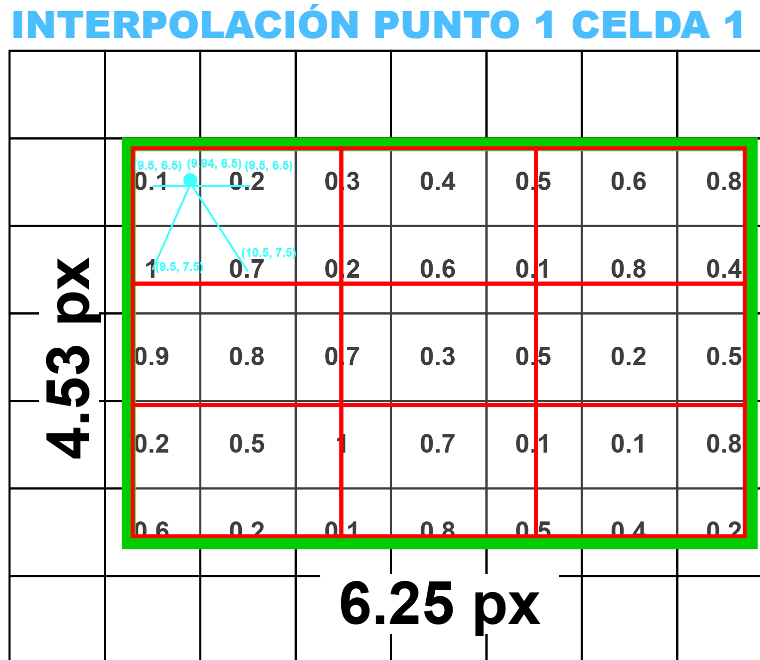



*Figura X. Interpolación de la ROI en punto 1 celda 1*

Sabiendo ya esto se puede calcular P para el punto 1:

$$P_1 \approx \frac{7.5-6.5}{7.5-6.5}\left(\frac{10.5-9.94}{10.5-9.5}\cdot 0.1+\frac{9.94-9.5}{10.5-9.5}\cdot 0.2\right)+$$

$$+\frac{6.5-6.5}{7.5-6.5}\left(\frac{10.5-9.94}{10.5-9.5}\cdot 1+\frac{9.94-9.5}{10.5-9.5}\cdot 0.7\right)\approx 0.14.$$

Ahora el procedimiento es el mismo para el resto de punto.

**INTERPOLACIÓN PUNTO 2 CELDA 1:**

- Vecinos del punto 2:

- Arriba a la izquierda: (10.5,6.5)
- Abajo a la izquierda: (10.5,7.5)
- Arriba a la derecha: (11.5,6.5)
- Abajo a la derecha: (11.5,7.5)

En la siguiente figura se ilustra al punto 2 que se va a interpolar junto a sus vecinos:

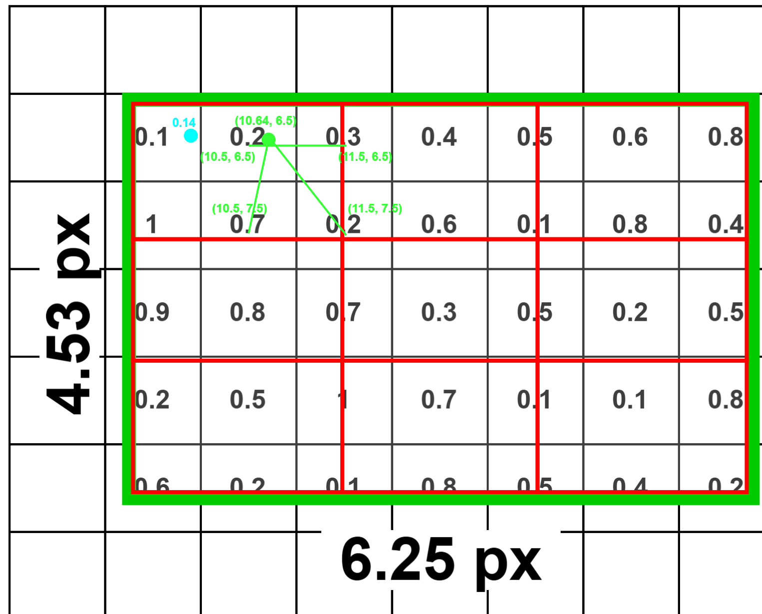



*Figura X. Interpolación de la ROI en punto 2 celda 1*

Cálculo de P para el punto 2:

$$P_2 \approx \frac{7.5-6.5}{7.5-6.5}\left(\frac{11.5-10.64}{11.5-10.5}\cdot 0.2+\frac{10.64-10.5}{11.5-10.5}\cdot 0.3\right)+$$

$$+\frac{6.5-6.5}{7.5-6.5}\left(\frac{11.5-10.64}{11.5-10.5}\cdot 0.7+\frac{10.64-10.5}{11.5-10.5}\cdot 0.2\right)\approx 0.21.$$

**INTERPOLACIÓN PUNTO 3 CELDA 1:**

- Vecinos del punto 3:

- Arriba a la izquierda: (9.5,6.5)
- Abajo a la izquierda: (9.5,7.5)
- Arriba a la derecha: (10.5,6.5)
- Abajo a la derecha: (10.5,7.5)

En la siguiente figura se ilustra al punto 3 que se va a interpolar junto a sus vecinos:

**INTERPOLACIÓN PUNTO 1 CELDA 3**

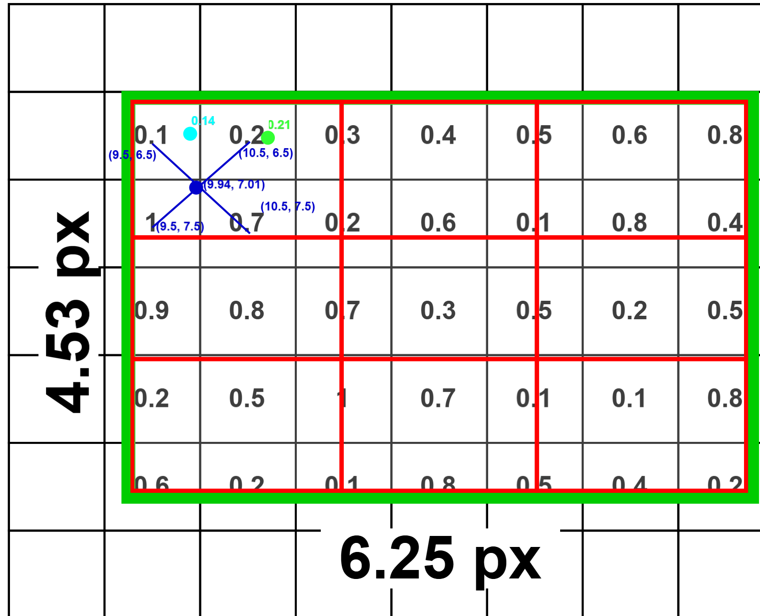



*Figura X. Interpolación de la ROI en punto 3 celda 1*

Cálculo de P para el punto 3:

$$P_3 \approx \frac{7.5-7.01}{7.5-6.5}\left(\frac{10.5-9.94}{10.5-9.5}\cdot 0.1+\frac{9.94-9.5}{10.5-9.5}\cdot 0.2\right)+$$

$$+\frac{7.01-6.5}{7.5-6.5}\left(\frac{10.5-9.94}{10.5-9.5}\cdot 1+\frac{9.94-9.5}{10.5-9.5}\cdot 0.7\right)\approx 0.51.$$

**INTERPOLACIÓN PUNTO 4 CELDA 1:**

- Vecinos del punto 4:

- Arriba a la izquierda: (10.5,6.5)
- Abajo a la izquierda: (10.5,7.5)
- Arriba a la derecha: (11.5,6.5)
- Abajo a la derecha: (11.5,7.5)

En la siguiente figura se ilustra al punto 4 que se va a interpolar junto a sus vecinos:

**INTERPOLACIÓN PUNTO 1 CELDA 4**

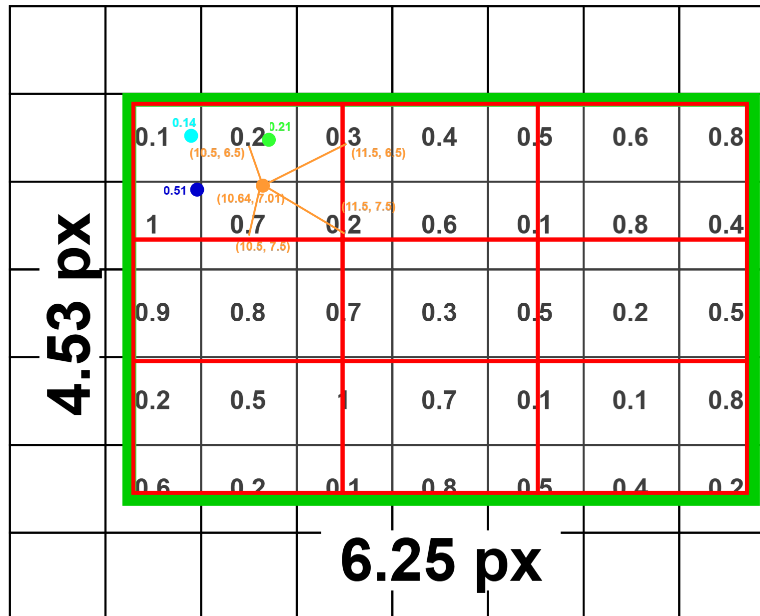



*Figura X. Interpolación de la ROI en punto 4 celda 1*

Cálculo de P para el punto 4:

$$P_4 \approx \frac{7.5-7.01}{7.5-6.5}\left(\frac{11.5-10.64}{11.5-10.5}\cdot 0.2+\frac{10.64-10.5}{11.5-10.5}\cdot 0.3\right)+$$

$$+\frac{7.01-6.5}{7.5-6.5}\left(\frac{11.5-10.64}{11.5-10.5}\cdot 0.7+\frac{10.64-10.5}{11.5-10.5}\cdot 0.2\right)\approx 0.43$$

Ahora que se tienen calculados todos los puntos se puede aplicar *max pooling* sobre ellos para obtener cual es el verdadero valor calculado, el cual será el valor que se pondrá en la celda superior izquierda de la matriz 3x3 resultante de hace ROI *Align*:

Valor celda superior izquierda de la matriz:

Max(0.14,0.21,0.51,0.43)=0.51

Para obtener el resto de los valores de las celdas de la matriz 3x3 se repite el mismo procedimiento anterior pero pasando a la siguiente celda del ROI. Una vez realizado el procedimiento la matriz resultante de hacer ROI *Align* a la ROI original es:

| | | |
|---|---|---|
| 0.51 | 0.46 | 0.71 |
| 0.86 | 0.50 | 0.52 |
| 0.56 | 0.83 | 0.30 |