Universidad Nacional
de Educación a Distancia

Escuela Técnica Superior de
Ingeniería Informática

Universidad Complutense de
Madrid

Facultad de Informática

# Detección de flejes en bobinas de acero mediante redes neuronales convolucionales.

Alberto Fuster González

Director: Gonzalo Pajares Martinsanz

Trabajo de Fin de Máster

Máster Universitario
en Automatización y Control Industrial

Junio 2025

# Agradecimientos

A mi esposa e hijos, por su paciencia, comprensión y apoyo incondicional durante el desarrollo de este Trabajo de Fin de Máster.

# Resumen

Este proyecto tiene como objetivo la detección de objetos en entornos industriales (concretamente la detección de flejes en bobinas de acero) mediante técnicas de *Deep Learning*. Para ello, se hace uso de redes neuronales convolucionales, presentando una introducción a su funcionamiento, a los principales hiperparámetros que influyen en el aprendizaje y a las métricas empleadas para evaluar su rendimiento. Se ha trabajado específicamente con el modelo YOLOv8, ampliamente utilizado en la detección de objetos.

Dado que no se dispone de un conjunto de datos amplio, se ha optado por aplicar técnicas de transferencia de aprendizaje, utilizando modelos preentrenados que ya han aprendido representaciones útiles a partir de grandes volúmenes de datos. Esto permite reutilizar las capas iniciales del modelo, que extraen características generales de las imágenes, y ajustar únicamente las capas finales para adaptarlas a la tarea específica de detección de los objetos industriales. Además, se han realizado pruebas con distintos tamaños de modelo y configuraciones de hiperparámetros, con el objetivo de evaluar su impacto en el rendimiento final.

Se aplican también conceptos de visión por computador, como la transformación afín y en perspectiva de la imagen.

Los recursos utilizados son un ordenador personal con 8Gb de memoria RAM, un procesador INTEL i7-7700 y una GPU Nvidia GTX1050Ti 4Gb con soporte CUDA. Para el entrenamiento de los modelos se ha utilizado la GPU y se ha limitado el tamaño de los lotes (batch) al máximo que permitía usar solo memoria de la GPU.

En lo referente al software, se ha realizado en Python y se han utilizado las librerías de openCV, Pytorch, Ultralytics, Numpy. Como editor de código fuente, se ha utilizado Visual Studio Code.

Palabras: Aprendizaje profundo, redes neuronales, YOLO.

# *Abstract*

*This project aims to detect objects in industrial environments—specifically, the presence of strapping bands on steel coils—using Deep Learning techniques. To achieve this, convolutional neural networks are employed, with an introduction to their functioning, the main hyperparameters that influence learning, and the metrics used to evaluate performance. The work focuses specifically on the YOLOv8 model, which is widely used in object detection tasks.*

*Due to the limited size of the available dataset, transfer learning techniques have been applied by using pretrained models that have already learned useful representations from large volumes of data. This allows the reuse of the initial layers of the model, which extract general features from images, while only the final layers are fine-tuned to adapt the network to the specific task of detecting industrial objects. Additionally, experiments have been conducted using different model sizes and hyperparameter configurations to evaluate their impact on overall performance.*

*The resources used are a personal computer with 8Gb of RAM memory, an INTEL i7-7700 processor, and an Nvidia GTX1050Ti 4Gb GPU with CUDA support. For model training, the GPU was used, and the batch size was limited to the maximum allowed by the GPU memory.*

*Regarding software, the project was developed in Python, and the openCV, Pytorch, Ultralytics, and Numpy libraries were used. Visual Studio Code was used as the source code editor.*

*Keywords: Deep learing, Convolutional Neuralnetwork, YOLO.*

# Índice general

# Índice de Figuras

# Índice de Tablas

# Capítulo 1

# Introducción

## 1.1. Motivación

La motivación de este proyecto estriba en la utilización de redes neuronales convolucionales para un uso industrial aplicado en la detección de flejes en bobinas de acero. El flejado es el proceso por el que mediante una cinta metálica o de plástico se envuelve la bobina evitando que las espiras exteriores se desplacen durante su manipulación.

Por lo que, tras cada proceso siderúrgico de decapado, laminado, recocido, electrozincado, etc., la bobina debe ser flejada para su manipulación con el puente grúa y almacenaje hasta el siguiente proceso. Es un requisito para transportar la bobina dentro de los procesos industriales de la propia empresa siderúrgica y para expedirla a cliente. Mover una bobina sin fleje supone un riesgo para la seguridad de las personas y de los equipos industriales.

Dado que en las instalaciones industriales y almacenes de bobinas normalmente hay instaladas cámaras de red para el control del proceso, la idea de este proyecto es demostrar que pueden usarse las imágenes obtenidas de estas cámaras para detectar la presencia de flejes en las bobinas usando redes neuronales convoluciones.

Existen aplicaciones comerciales para detección de defectos en la superficie de la bobina, con fabricantes como COGNEX e ISRA VISION. Existen también sistemas que montan sobre un robot multibrazo un telemetro láser y un cámara para localizar el fleje y quitarlo antes del siguiente proceso, con fabricantes como SIGNODE o TEBULO ROBOTICS. Pero no hay ninguna solución comercial que verifique que la bobina lleva los flejes y que estos están en la posición correcta.

Se utiliza YOLOv8 [1], que es un sistema de detección de objetos en tiempo real que utiliza aprendizaje profundo para detectar objetos en imágenes y videos. Es un método muy popular en el campo de la visión por computadora y está suficientemente documentado, de ahí que se haya seleccionado para este proyecto. Se trata de un modelo de un solo estado, de forma que a diferencia de los de dos estados, como los del tipo R-CNN, que se mencionan después, la imagen se procesa en una única pasada. En los de dos estados, se realiza una propuesta inicial de regiones candidatas para después procesarlas con vistas a la detección de objetos.

Utiliza una sola red neuronal en un solo paso para predecir cajas delimitadoras y probabilidades de clase directamente desde imágenes completas. Destaca por tener una velocidad de inferencia muy rápida. Dispone de 5 versiones escaladas: YOLOv8n (*nano*), YOLOv8s (*small*), YOLOv8m (*medium*), YOLOv8l (*large*) and YOLOv8x (*extra large*). En la Figura 1 se muestran sendas gráficas ilustrativas del rendimiento de varias versiones de YOLO sobre el conjunto de datos COCO. La imagen de la Figura 1 izquierda, muestra la relación entre el número de parámetros y precisión en la detección usando la métrica COCO mAP50-95. La gráfica de la derecha muestra la relación entre velocidad de inferencia y precisión

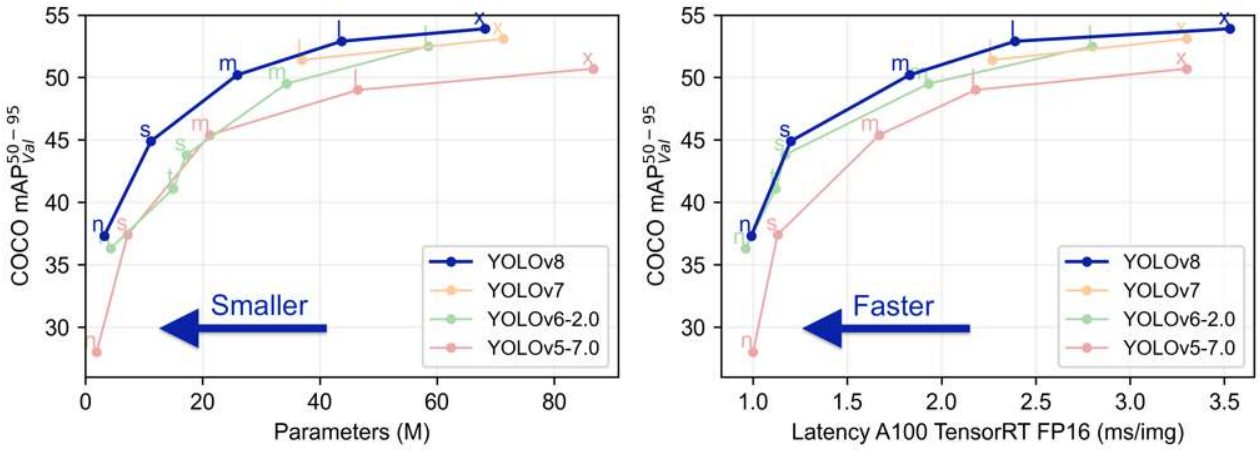



*Figura 1: Diferentes modelos de detección de objetos de YOLOV8.. Gráfica obtenida de [2].*

Otras alternativas a YOLOV8 pueden ser:

- R-CNN (*Region-based Convolutional Neural Networks*)[4], y sus sucesores dentro de la misma familia Fast R-CNN [5] y Faster R-CNN [6]: son métodos de detección de objetos de dos estados con tiempos de inferencia muy superiores. Utiliza una red de propuestas de regiones (RPN) para generar propuestas y luego clasificarlas utilizando una CNN.

- SSD (Detector de un solo disparo): SSD [3] es un método de detección de objetos de un estado en tiempo real que utiliza una sola red neuronal para predecir localizaciones de objetos y clases. Realiza predicciones directamente desde múltiples mapas de características de diferentes resoluciones.

## 1.2. Propuesta y objetivos

El objetivo es demostrar que pueden usarse métodos de aprendizaje profundo y, mediante transferencia de aprendizaje, entrenar el modelo con un conjunto de datos pequeño, en este caso de un entorno industrial.

Normalmente en entornos industriales va a ver poca variabilidad en las imágenes y la posición de las cámaras será fija, habitualmente no será necesario identificar muchas clases de objetos. Con un modelo entrenado con un *dataset* tan grande como COCO, el modelo ha aprendido a distinguir muchas características de la imagen. Entrenar el modelo con el conjunto de datos objetivo permitirá ajustar las últimas capas de la red neuronal para localizar e identificar las distintas clases.

Se identificará cómo afecta la elección de la complejidad del modelo al sobreajuste durante el entrenamiento y a las métricas, así como la influencia de diferentes hiperparámetros en el entrenamiento.

Otro objetivo del proyecto es realizar transformaciones geométricas de la imagen, que permitan tener vistas de la bobina que quizás no puedan obtenerse con las cámaras de red instaladas.

Se identificarán los flejes circunferenciales y radiales, localizando además la posición angular de los flejes radiales.

## 1.3. Estructura del documento

La presente memoria se estructura en capítulos de la siguiente manera:

1. El capítulo uno, contiene la introducción, donde se expone la motivación del trabajo y las tareas realizas, así como la definición de los objetivos.

2. El segundo capítulo incluye aspectos teóricos sobre redes neuronales convolucionales, métricas utilizadas para detección de objetos. Se introduce también el algoritmo detección en un solo paso YOLO.

3. En el tercer capítulo se describen los recursos utilizados, tanto a nivel de *hardware* como de *software* así como los métodos utilizados.

4. En el cuarto capítulo se presentan los resultados, utilizando principalmente la métrica COCO mAP50-95. También se comparan tiempos de entrenamiento para diferentes modelos, y pruebas con diferentes hiperparámetros.

5. Finalmente, en el quinto capítulo de exponen las conclusiones y trabajo futuro.

# Capítulo 2

# Estado del Arte

El estado del arte en detección de objetos en visión artificial utilizando redes neuronales convolucionales (CNN, *Convolutional Neural Networks*) ha experimentado un rápido avance en los últimos años. Las CNN han demostrado ser efectivas en la detección de objetos en imágenes y videos, superando a los métodos tradicionales basados en características.

Entre los enfoques más populares se encuentran los métodos de detección de objetos basados en regiones, como RCNN, Fast RCNN y Faster RCNN. Estos detectores, denominados de dos etapas, tienen un alto coste computacional dado que las regiones de interés (ROI) pasan por una CNN preentrenada, de ahí el nombre de dos etapas, ya que en primer lugar se hace una propuesta de regiones para proceder posteriormente a identificar el objeto Son además más complejos de entrenar.

Otros enfoques, como YOLO (*You Only Look Once*) y SSD (*Single Shot Multibox Detector*), utilizan una sola red neuronal convolucional para detectar objetos en una sola etapa, sin necesidad de generar propuestas de regiones. Estos métodos que originalmente resultaban menos precisos que los anteriores son sin embargo mucho más rápidos, pues consiguen en una sola etapa generar la región y clasificar el objeto.

Las CNN han demostrado ser capaces de aprender características robustas para la detección de objetos, lo que les permite generalizar bien a nuevos conjuntos de datos y entornos. Además, la capacidad de las CNN para procesar grandes cantidades de datos y aprender de ellos ha permitido la creación de modelos de detección de objetos cada vez más precisos y robustos.

No obstante, aún existen desafíos importantes en la detección de objetos, como la detección de objetos pequeños o parcialmente ocultos, la variabilidad en la iluminación y la presencia de ruido en las imágenes, etc.

## 2.1. Introducción a las redes neuronales convolucionales

Las CNN son un tipo de red neuronal artificial que se ha vuelto esencial en el campo de la visión artificial. Aunque comparten algunas similitudes con el perceptrón multicapa, las CNN tienen una arquitectura y un funcionamiento únicos que las hacen especialmente adecuadas para tareas de procesamiento de imágenes, eso se debe precisamente al uso de la conocida operación de convolución.

### 2.1.1 Comparación con el Perceptrón Multicapa

El perceptrón multicapa (en adelante MLP: *Multilayer Perceptron*) es un tipo de red neuronal artificial que consiste en varias capas de neuronas interconectadas. Cada capa recibe la salida de la capa anterior y la procesa mediante una función de activación no lineal. Es capaz de aprender patrones complejos en los datos, pero tiene algunas limitaciones cuando se trata de procesar imágenes, por ejemplo, una oclusión en la imagen, una rotación o una traslación de la imagen, así como un cambio de brillo modificarán sensiblemente el desempeño.

Las imágenes son matrices de píxeles que contienen información espacial y estructural. El MLP no es tan eficaz como la CNN a la hora de capturar esta información de manera efectiva, ya que procesa cada píxel de manera independiente sin preservar la estructura espacial.

Por otro lado, las CNN están diseñadas específicamente para procesar imágenes y capturar la información espacial y estructural que contienen. Realiza convoluciones recorriendo toda la imagen, de tal forma que extrae sus características sin importar la ubicación dentro de la imagen. Estas convoluciones alimentan a la siguiente convolución, generando características de la imagen cada vez más complejas.

Se muestra en la Figura 2 el esquema de una CNN para identificar caracteres numéricos con 2 capas convolucionales, 2 capas *maxPooling* y 2 capas completamente conectas y salida *softmax*.

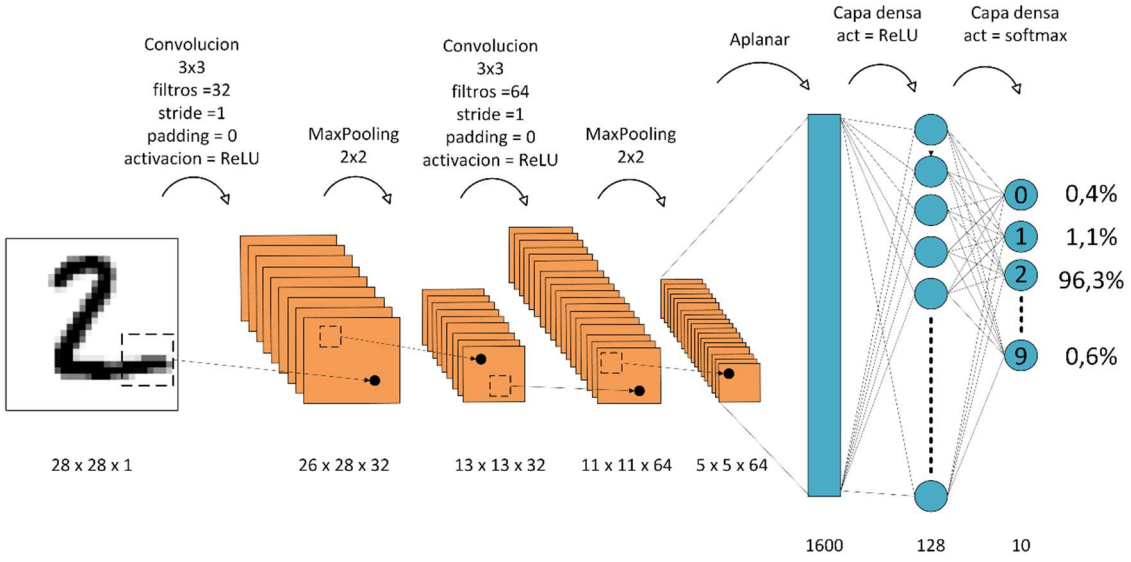



*Figura 2: Esquema de una red neuronal convolucional para identificar caracteres numéricos.*

### 2.1.2 Capas típicas de una Red Neuronal Convolucional:

Una CNN típica consiste en varias capas que trabajan juntas para procesar las imágenes. A continuación, se describen las diferentes capas que participan en la composición de una CNN con carácter general:

- **Capa de Entrada:** recibe la imagen de entrada y la convierte en un tensor que puede ser procesado por la red neuronal. Por ejemplo, en la Figura 2 la entrada es una imagen con una resolución de 28x28 y un solo canal en escala de gris, pero la entrada podría ser una imagen de color con tres canales. La imagen se representa en este caso como una matriz de píxeles con valores de intensidad que van desde 0 (negro) hasta 255 (blanco).

- **Capa de Convolución:** es la capa más importante de una CNN. En esta capa, se aplican filtros de convolución a la imagen de entrada para extraer características relevantes. Los filtros de convolución son matrices de coeficientes que se deslizan sobre la imagen de entrada, realizando una convolución en cada posición. La salida de la capa de convolución es un conjunto de mapas de características que representan la presencia de patrones en la imagen.

  A la primera capa de convolución se le suele llamar '***Stem***' y su misión es reducir el tamaño de entrada realizando una convolución de diferentes tamaños, según las dimensiones de los núcleos de convolución utilizados por ejemplo 5x5 ó 7x7 y salto cada varios pixeles. Esto permite extraer las características de la imagen sin usar una dimensión muy alta en el resto de capas.

- **Capa de Activación:** se aplica después de la capa de convolución y tiene como objetivo introducir no linealidad en la salida de la capa de convolución. La función de activación más común utilizada en las CNN es la función ReLU (*Rectified Linear Unit*), que asigna un valor de 0 a todos los valores negativos y deja intactos los valores positivos. Viene a ser algo tan sencillo como: max(0,X).

  En los inicios se usaba la función *sigmoide* pero daba problemas porque cuando la salida era -1 o 1 se congelaba el aprendizaje porque la derivada se hacía cero y no había actualización de parámetros en el proceso de *backpropagation*.

- **Capa de *Pooling*:** se utiliza para reducir la dimensión espacial de los mapas de características y mantener solo la información más relevante. La capa de *Pooling* se aplica después de la capa de activación y se utiliza para reducir la cantidad de parámetros y la complejidad computacional de la red neuronal. La técnica más habitual es la de quedarse con el valor más alto.

- **Capa de *Flattened*:** se utiliza para transformar la salida de la capa de *Pooling* en un vector unidimensional que pueda ser procesado por la capa totalmente conectada. Simplemente transforma un tensor tridimensional (alto, ancho y profundidad), en un vector unidimensional.

  En las redes neuronales más modernas se ha sustituido el *Flattened* por un *Average Pooling*. Esto tiene varias ventajas: reduce el número de parámetros y es más eficiente computacionalmente. Permite un tamaño variable de la imagen, porque se reduce cada canal a su media, obtenido una sola característica por canal. No obstante, se pierde información espacial de la imagen.

- **Capa Totalmente Conectada (FC, *Fully Connected*):** es la capa final de la CNN y se utiliza para realizar la clasificación o la regresión. En esta capa, se conectan todos los mapas de características de la capa de *Pooling* con una capa de neuronas que realiza la predicción final.

  En lugar de utilizar una capa FC, se puede utilizar una capa de *Pooling* global para reducir la dimensionalidad de la salida de la capa de convolución. Luego, se puede agregar una capa de *softmax* para producir la salida final.

### 2.1.3 Operaciones en una Red Neuronal Convolucional:

**Función de activación:** es una función matemática utilizada para introducir no linealidad, la función de activación se aplica a la salida de cada neurona y permitir que la red aprenda patrones más complejos.

Las funciones de activación más comunes son:

***a) Sigmoid*:** $\sigma(x) = \frac{1}{(1+\exp(-x))}$ (2.1)

Muy utilizada en las primeras redes neuronales, si bien su problema de saturación hizo que se reemplazase por ReLU. La función, representada en la Figura 3, tiene una salida que se aproxima a 0 o 1 cuando la entrada es grande en valor absoluto. Por tanto, la pendiente de la curva se vuelve muy pequeña. Como resultado, el gradiente de la función de pérdida se torna muy pequeño durante el entrenamiento, lo que hace que el aprendizaje sea lento o incluso se detenga.

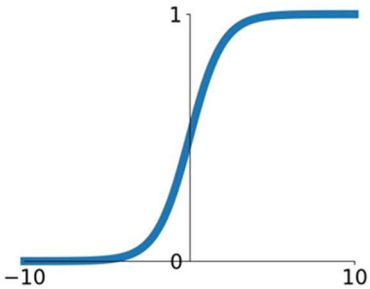



*Figura 3: Representación función de activación Sigmoide.*

***b)* ReLU (*Rectified Linear Unit*):** $f(x) = max(0, x)$ (2.2)

Es sin duda el más utilizado. ReLU, cuya representación se muestra en la Figura 4, ofrece varias ventajas sobre la función *sigmoide*: no saturación, lo que permite un aprendizaje más rápido y estable; computación más rápida, ya que no requiere la evaluación de la función exponencial; los gradientes no se desvanecen; centro en cero, lo que simplifica la normalización; no linealidad más fuerte, lo que permite modelar relaciones más complejas. Esto conduce a modelos más precisos y robustos.

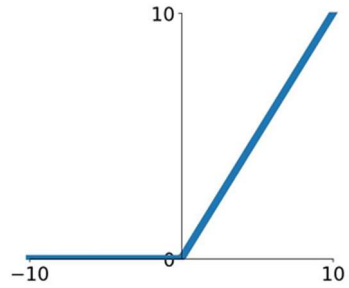



*Figura 4: Representación función de activación ReLU (Rectified Linear Unit).*

***c) Tanh (Hyperbolic Tangent):*** $f(x) = \frac{2}{1+\exp(-2x)} - 1$ (2.3)

En general, Tanh es una mejor opción que sigmoide en la mayoría de los casos, Figura 5, pero ReLU y sus variantes siguen siendo las funciones de activación más populares. Está centrada en cero pero tiene problemas de gradientes más pequeños en los extremos, lo que puede ralentizar el aprendizaje.

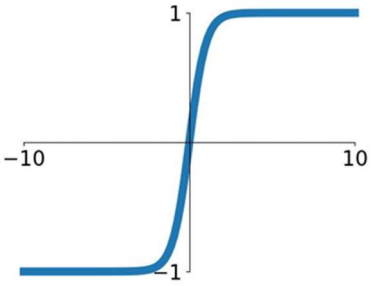



*Figura 5: Representación función de activación Tangente Hiperbólica.*

***d) Leaky ReLU:*** $f(x) = max(\alpha x, x)$, donde α es un parámetro que controla la cantidad de "*leakage*" cuyo valor irá entre 0.01 y 0.1

Es una mejora sobre ReLU, donde las neuronas que tienen una entrada negativa se "mueren" porque la salida es siempre 0. En *Leaky ReLU*, representada en la Figura 6, una pequeña fracción de la entrada se permite pasar, lo que evita que las neuronas se "mueran" y permite que sigan aprendiendo.

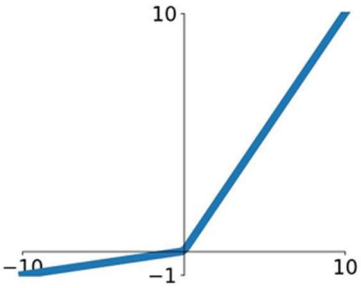



*Figura 6: Representación función de activación Leaky ReLU.*

**Normalización:** Es una técnica utilizada para estandarizar las entradas de una red neuronal. El objetivo es que las entradas tengan una media cercana a cero y una varianza cercana a uno. Esto ayuda a acelerar el entrenamiento, mejorar la convergencia y reducir el sobreajuste.

Existen diferentes técnicas de normalización:

- Normalización por lotes (***Batch Normalization***): se aplica a cada lote de entradas (batch) que se procesa en la red neuronal. El objetivo es normalizar las entradas de cada lote. Esto se logra calculando la media y la varianza de cada lote y luego normalizando las entradas en función de estos valores. Es la más utilizada.

- Normalización por instancia (***Instance Normalization***): se aplica a cada instancia individual de las entradas, es decir, a cada ejemplo de entrenamiento.

- Normalización por canal (***Channel Normalization***): se aplica a cada canal de las entradas, es decir, a cada característica o atributo de las entradas.

La Figura 7 muestra los distintos métodos de normalización. Cada subparcela muestra un tensor del mapa de características, con N como eje de lote, C como eje de canal y (H×W) como ejes espaciales. Los píxeles en azul están normalizados por la misma media y varianza, calculadas agregando los valores de estos píxeles.

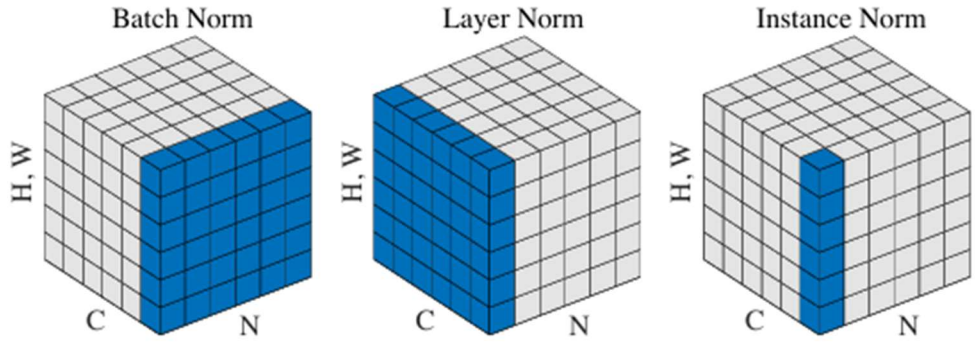



*Figura 7: Métodos de normalización. Representación gráfica tomada de [17]*

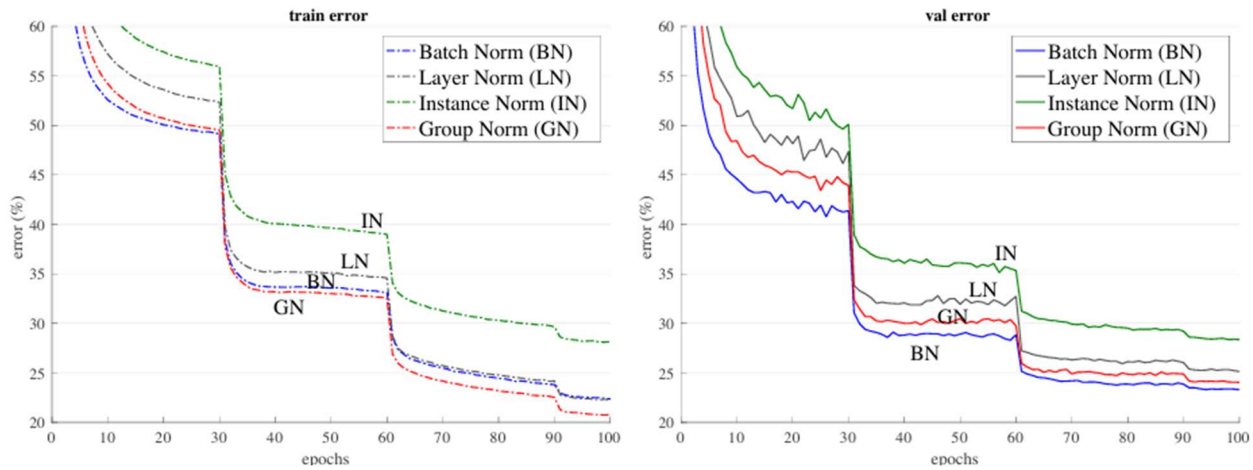



*Figura 8: Curvas de error con distintos métodos de normalización. Representación grafica tomad de [18]*

La Figura 8 muestra la comparación de curvas de error con un tamaño de lote de 32 imágenes obtenido por Wu and He en publicación "*Group Normalization*" [18]. Se muestra el error de entrenamiento (izquierda) y el error de validación (derecha) versus el número de veces de entrenamiento. El modelo utilizado en este trabajo se ResNet-50 y el *dataset* ImageNet.

**Sobreajuste (*Overfitting*):** El sobreajuste ocurre cuando una red neuronal se ajusta demasiado bien a los datos de entrenamiento y no generaliza bien a nuevos datos. Esto puede ocurrir cuando la red tiene demasiados parámetros o cuando se entrena durante demasiado tiempo. El sobreajuste se puede detectar mediante la curva de aprendizaje, que muestra la precisión en los datos de entrenamiento y los de validación.

**Regularización:** La regularización es una técnica utilizada para prevenir el sobreajuste en las redes neuronales. El objetivo es agregar un término de penalización al costo de la red para desalentar la complejidad excesiva del modelo. Algunas técnicas de regularización comunes son:

- ***Dropout*:** Es una técnica de regularización utilizada para prevenir el sobreajuste en las redes neuronales. Consiste en desactivar aleatoriamente una fracción de las neuronas durante el entrenamiento, lo que fuerza a la red a aprender múltiples representaciones de los datos y a no depender demasiado de una sola neurona. De esta manera, la red se vuelve más robusta y generaliza mejor a nuevos datos.

  En la Figura 9 se representa a la izquierda una red neuronal estándar con dos capas ocultas, en la derecha la misma red con un *dropout* del 0,5. La mitad de los parámetros de la red no se entrenan en el número de *epoch* correspondiente.

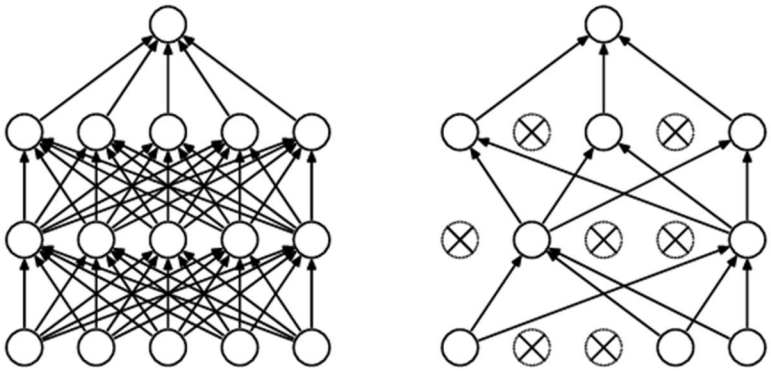



*Figura 9: Ejemplo de dropout = 0,5. Representación gráfica tomada de [19]*

- ***L1 Regularization (Lasso)*:** se agrega un término de penalización proporcional a la suma de los valores absolutos de los pesos. Donde $\lambda$ será un hiperparámetro.

$$Loss = Data\ Loss + \lambda\, R(W) \quad (2.4)$$
$$R(W) = \sum_k \sum_l \left|W_{k,l}\right| \quad (2.5)$$

- ***L2 Regularization (Ridge)*:** se agrega un término de penalización proporcional a la suma de los cuadrados de los pesos. Donde $\lambda$ será un hiperparámetro.

$$Loss = Data\ Loss + \lambda\, R(W)$$

$$R(W) = \sum_k \sum_l W_{k,l}^2 \qquad (2.6)$$

- ***Early Stopping*:** se detiene el entrenamiento cuando la precisión en la validación comienza a disminuir

***Softmax*:** La función *softmax* es una función de activación utilizada comúnmente en la capa de salida de una red neuronal cuando se necesita producir una distribución de probabilidad sobre varias clases (más de dos clases).

Se define como: $softmax(x) = \frac{exp\,(x)}{\Sigma\, exp(x)}$, donde x es el vector de entradas y Σ exp(x) es la suma de los exponentes de las entradas.

La función *softmax* garantiza que la salida sea una distribución de probabilidad válida, es decir, que la suma de las probabilidades sea igual a 1.

**Función de pérdida:** Mide la diferencia entre las predicciones del modelo y las etiquetas de verdad. Se utiliza su gradiente para ajustar los pesos de los parámetros de la red neuronal durante la retropropagación (*backpropagation*). Es habitual usar la denominada *Cateogorical Cross-Entropy* cuando hay más de dos clases.

Obtenemos la función de pérdida *Categorical Cross-Entropy* a partir de la función *Softmax* como el valor negativo del logaritmo de la predicción de la clase correcta 'i' en relación al resto de categorías:

$$Loss = -\log\left(\frac{\exp(x_i)}{\Sigma \exp(x)}\right) \qquad (2.7)$$

***Stride:*** Define el número de píxeles que se desplaza el *kernel* al aplicarlo sobre la imagen si es la primera capa o sobre el campo de activaciones en el resto de las capas intermedias. Un *stride* de 1 significa que el *kernel* se desplaza píxel a píxel al aplicar la operación de convolución, para reducir el tamaño de salida se usa un *stride* mayor a 1, de tal modo que la operación de convolución realiza saltos cuyo valor es el *stride*.

***Padding:*** Consiste en añadir ceros en los bordes de la imagen o mapa de activaciones para preservar el tamaño tras las operaciones de convolución.

### 2.1.4 Evolución de las CNN

Entre 2012 y 2016, las redes neuronales experimentaron un gran avance en el campo del reconocimiento de imágenes. En 2012, el equipo de Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton ganó el *ImageNet Large Scale Visual Recognition Challenge* con una precisión del 15.3% utilizando una red neuronal convolucional (CNN) llamada **AlexNet** [9]. Fue una de las primeras en demostrar la efectividad de las CNNs en el reconocimiento de imágenes frente a métodos anteriores que utilizaban métodos clásicos de visión por computador. Sentó las bases para el desarrollo de redes neuronales convolucionales más avanzadas.

La Figura 10 muestra los ganadores del *ImageNet Large Scale Visual Recognition Challenge*, donde AlexNet [9] fue la primera red neuronal convolucional en ganar este desafío y mejorar sensiblemente los resultados obtenidos anteriormente con métodos basados en extracción de características de la imagen: transformada de características invariantes (SIFT) y vectores

Fisher (FVs).

A partir de ese momento todos los ganadores usarían redes neuronales convolucionales (CNN). Puede observarse también cómo evolucionaría la profundidad de la redes neuronales, aumentando el número de capas.

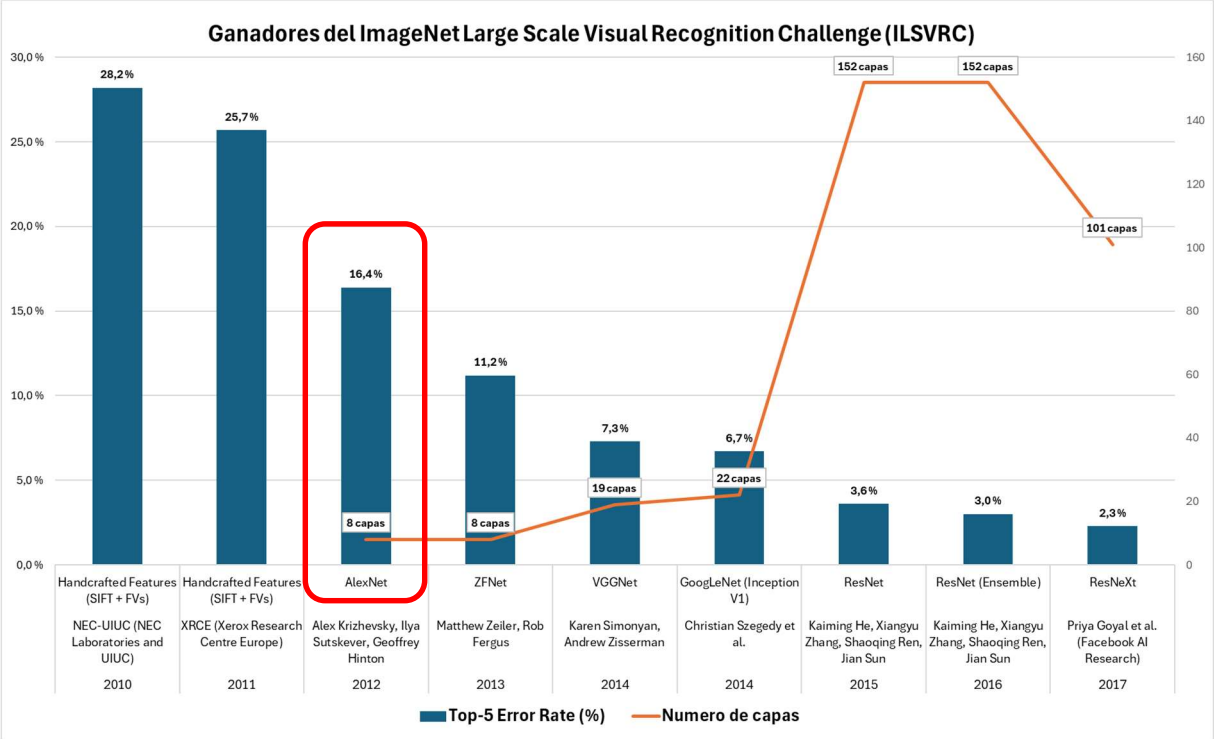



*Figura 10: Ganadores del ILSVRC*

La red AlexNet es considerada una de las redes neuronales convolutionales (CNNs) más influyentes y pioneras en el campo del aprendizaje automático y visión por computadora, constituyendo un punto de inflexión, dado que superaría a otros modelos de aprendizaje automático clásicos y demostraría la efectividad de las CNNs en el reconocimiento de imágenes.

La red AlexNet introdujo varias innovaciones clave, como:

- La utilización de unidades de procesamiento gráfico (GPUs) para acelerar el entrenamiento de la red.
- La utilización de la función de activación ReLU.
- La utilización de técnicas de regularización, como *dropout* y *weight decay*, para prevenir el *overfitting*. La tasa de aprendizaje utilizada es de 1E-2, reducida por 10 manualmente cuando la presión en los datos de validación ya no mejora en los sucesivos *epochs*.
- La utilización de una arquitectura de CNN más profunda y ancha que las redes anteriores.
- La utilización de varias capas de convolución apiladas antes del *pooling*.

En la Figura 11 se muestra la estructura de la AlexNet tal y como se publicó en el trabajo original, hay un pequeño error porque el tamaño de la imagen de entrada debería ser 227×227×3.

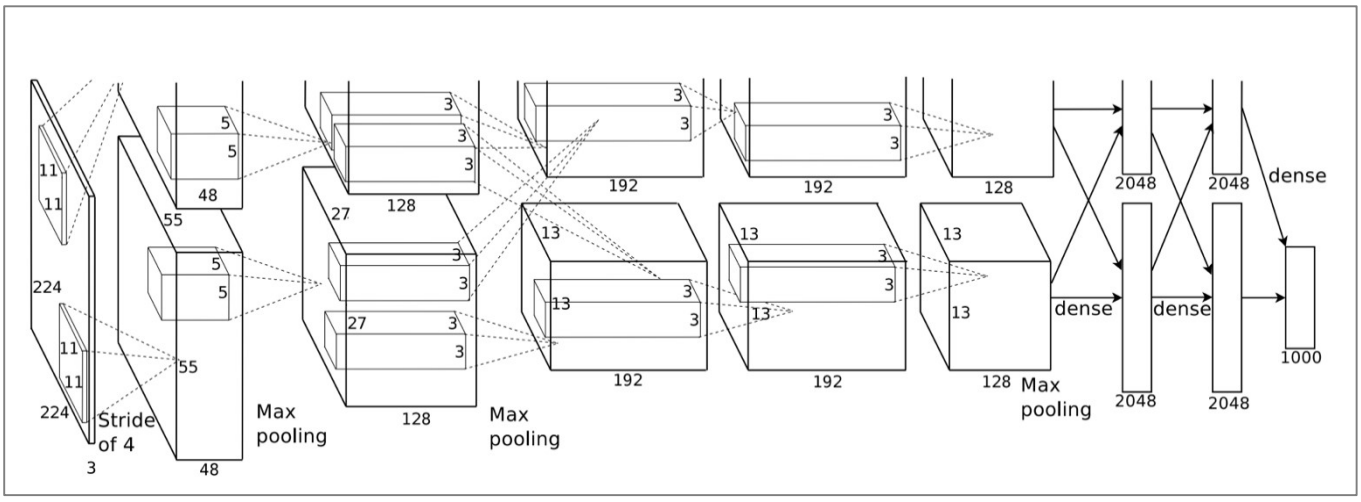



*Figura 11: Estructura AlexNet. Ilustración gráfica tomada de [9]*

El éxito de la red AlexNet llevó a un aumento en la popularidad de las CNNs y sentó las bases para el desarrollo de redes neuronales convolutionales más avanzadas, como la red VGGNet [10], la red GoogLeNet [11] y la red ResNet [12].

En el cuadro mostrado en la Tabla 1, pueden verse las distintas capas, el *kernel*, *stride*, *padding* y número de filtros. Es interesante ver el número de parámetros que maneja esta red, unos 62 millones, aproximadamente 238Mb (almacenados en un float32), lo cual supone que en cada *backpropagation* tiene que almacenar en memoria de la GPU el valor de este parámetro y su gradiente, junto con los valores de activación y sus gradientes. Además, si este proceso se hace en un *batch* de por ejemplo 8 imágenes, todo esto multiplicado por ese valor de *batch.*

En la Tabla 1, se muestran las entradas y salidas de cada capa, el *kernel* utilizado y el número de filtros. El número de parámetros, aproximadamente 62 millones y el tamaño de activaciones (salidas de cada capa después de la función de activación) de aprox. 780.000.

*Tabla 1: Arquitectura de **AlexNet** [9]*

| | | | | | | | | | | | | | float32 data type | | | float32 data type | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **ALEXNET** | | | | | | | | | | | | | MB | | | MB | |
| **CAPA** | **ENTRADA** | | | **SALIDA** | | | **KERNEL** | | **STRIDE** | **PAD** | **Nº FILTROS** | **PARAMETROS** | **MEMORIA PARAMETROS** | **%** | **ACTIVACIO-NES** | **MEMORIA ACTIVACIONES** | **%** |
| CONV 1 | 227 | 227 | 3 | 55 | 55 | 96 | 11 | 11 | 4 | 0 | 96 | 34.944,00 | 0,133 | 0,1% | 290.400,00 | 1,108 | 37,1% |
| MAXPOOL 1 | 55 | 55 | 96 | 27 | 27 | 96 | 3 | 3 | 2 | 0 | | | - | | 69.984,00 | 0,267 | 9,0% |
| CONV 2 | 27 | 27 | 96 | 27 | 27 | 256 | 5 | 5 | 1 | 2 | 256 | 614.656,00 | 2,345 | 1,0% | 186.624,00 | 0,712 | 23,9% |
| MAXPOOL | 27 | 27 | 256 | 13 | 13 | 256 | 3 | 3 | 2 | 0 | | | - | | 43.264,00 | 0,165 | 5,5% |
| CONV 3 | 13 | 13 | 256 | 13 | 13 | 384 | 3 | 3 | 1 | 1 | 384 | 885.120,00 | 3,376 | 1,4% | 64.896,00 | 0,248 | 8,3% |
| CONV 4 | 13 | 13 | 384 | 13 | 13 | 384 | 3 | 3 | 1 | 1 | 384 | 1.327.488,00 | 5,064 | 2,1% | 64.896,00 | 0,248 | 8,3% |
| CONV 5 | 13 | 13 | 384 | 13 | 13 | 256 | 3 | 3 | 1 | 1 | 256 | 884.992,00 | 3,376 | 1,4% | 43.264,00 | 0,165 | 5,5% |
| MAXPOOL | 13 | 13 | 256 | 6 | 6 | 256 | 3 | 3 | 2 | 0 | | 0 | - | | 9.216,00 | 0,035 | 1,2% |
| FC6 | 9216 | | | 4096 | | | | | | | | 37.752.832,00 | 144,016 | 60,5% | 4.096,00 | 0,016 | 0,5% |
| FC7 | 4096 | | | 4096 | | | | | | | | 16.781.312,00 | 64,016 | 26,9% | 4.096,00 | 0,016 | 0,5% |
| FC8 | 4096 | | | 1000 | | | | | | | | 4.097.000,00 | 15,629 | 6,6% | 1.000,00 | 0,004 | 0,1% |
| | | | | | | | | | | | | **62.378.344,00** | **237,95** | **100%** | **781.736,00** | **2,98** | **100%** |

Es interesante observar cómo los mayores requerimientos de memoria en los parámetros están en las capas FC (perceptrón multicapa) y en cambio en la parte de los elementos de activación la mayor cantidad de memoria se requiere en las primeras capas. De ahí la importancia de un buen *stem network* (primera capa después de la imagen, normalmente con dimensiones de núcleo de

11x11 o 7x7) para reducir rápidamente el número de activaciones, y por tanto también el número de operaciones.

En 2014, el equipo de **VGGNet**[10] utilizó una arquitectura de red neuronal más profunda y alcanzó una precisión del 7.3% en la tarea de localización del top-5. VGGNet incrementó la profundidad utilizando simples filtros de 3x3, lo que le permitió capturan más características de la imagen, pero se incrementó sustancialmente el número de parámetros.

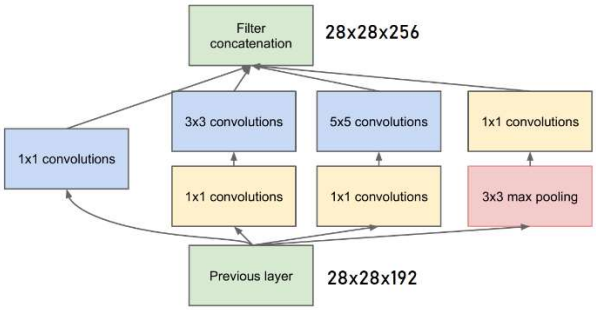
Tal y como puede observarse en la Tabla 2, VVG16[10] es una red más profunda, con 19 capas y tamaño de parámetros muy grande, 138 millones lo que supone 527Mb (en float32).

*Tabla 2: Arquitectura **VVG16**[10]*

| | | | | | | | | | | | | | float32 data type | | | float32 data type | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **VGG16** | | | | | | | | | | | | | MB | | | MB | |
| **CAPA** | **ENTRADA** | | | **SALIDA** | | | **KERNEL** | | **STRIDE** | **PAD** | **Nº FILTROS** | **PARAMETROS** | **MEMORIA PARAMETROS** | **%** | **ACTIVACIONES** | **MEMORIA ACTIVACIONES** | **%** |
| CONV 1-1 | 224 | 224 | 3 | 224 | 224 | 64 | 3 | 3 | 1 | 1 | 64 | 1.792,00 | 0,007 | 0,00% | 3.211.264,00 | 12,250 | 21,3% |
| CONV 1-2 | 224 | 224 | 64 | 224 | 224 | 64 | 3 | 3 | 1 | 1 | 64 | 36.928,00 | 0,141 | 0,03% | 3.211.264,00 | 12,250 | 21,3% |
| MAXPOOL 1 | 224 | 224 | 64 | 112 | 112 | 64 | 2 | 2 | 2 | 0 | | | - | 0,00% | 802.816,00 | 3,063 | 5,3% |
| CONV 2-1 | 112 | 112 | 64 | 112 | 112 | 128 | 3 | 3 | 1 | 1 | 128 | 73.856,00 | 0,282 | 0,05% | 1.605.632,00 | 6,125 | 10,6% |
| CONV 2-2 | 112 | 112 | 128 | 112 | 112 | 128 | 3 | 3 | 1 | 1 | 128 | 147.584,00 | 0,563 | 0,11% | 1.605.632,00 | 6,125 | 10,6% |
| MAXPOOL 2 | 112 | 112 | 128 | 56 | 56 | 128 | 2 | 2 | 2 | 0 | | | - | 0,00% | 401.408,00 | 1,531 | 2,7% |
| CON 3-1 | 56 | 56 | 128 | 56 | 56 | 256 | 3 | 3 | 1 | 1 | 256 | 295.168,00 | 1,126 | 0,21% | 802.816,00 | 3,063 | 5,3% |
| CON 3-2 | 56 | 56 | 256 | 56 | 56 | 256 | 3 | 3 | 1 | 1 | 256 | 590.080,00 | 2,251 | 0,43% | 802.816,00 | 3,063 | 5,3% |
| CON 3-3 | 56 | 56 | 256 | 56 | 56 | 256 | 3 | 3 | 1 | 1 | 256 | 590.080,00 | 2,251 | 0,43% | 802.816,00 | 3,063 | 5,3% |
| MAXPOOL 3 | 56 | 56 | 256 | 28 | 28 | 256 | 2 | 2 | 2 | 0 | | | - | 0,00% | 200.704,00 | 0,766 | 1,3% |
| CONV4-1 | 28 | 28 | 256 | 28 | 28 | 512 | 3 | 3 | 1 | 1 | 512 | 1.180.160,00 | 4,502 | 0,85% | 401.408,00 | 1,531 | 2,7% |
| CONV4-2 | 28 | 28 | 512 | 28 | 28 | 512 | 3 | 3 | 1 | 1 | 512 | 2.359.808,00 | 9,002 | 1,71% | 401.408,00 | 1,531 | 2,7% |
| CONV4-3 | 28 | 28 | 512 | 28 | 28 | 512 | 3 | 3 | 1 | 1 | 512 | 2.359.808,00 | 9,002 | 1,71% | 401.408,00 | 1,531 | 2,7% |
| MAXPOOL 4 | 28 | 28 | 512 | 14 | 14 | 512 | 2 | 2 | 2 | 0 | | | - | 0,00% | 100.352,00 | 0,383 | 0,7% |
| CONV5-1 | 14 | 14 | 512 | 14 | 14 | 512 | 3 | 3 | 1 | 1 | 512 | 2.359.808,00 | 9,002 | 1,71% | 100.352,00 | 0,383 | 0,7% |
| CONV5-2 | 14 | 14 | 512 | 14 | 14 | 512 | 3 | 3 | 1 | 1 | 512 | 2.359.808,00 | 9,002 | 1,71% | 100.352,00 | 0,383 | 0,7% |
| CONV5-3 | 14 | 14 | 512 | 14 | 14 | 512 | 3 | 3 | 1 | 1 | 512 | 2.359.808,00 | 9,002 | 1,71% | 100.352,00 | 0,383 | 0,7% |
| MAXPOOL 5 | 14 | 14 | 512 | 7 | 7 | 512 | 2 | 2 | 2 | 0 | | | - | 0,00% | 25.088,00 | 0,096 | 0,2% |
| FC1 | 25088 | | | 4096 | | | | | | | | 102.764.544,00 | 392,016 | 74,27% | 4096 | 0,016 | 0,0% |
| FC2 | 4096 | | | 4096 | | | | | | | | 16.781.312,00 | 64,016 | 12,13% | 4096 | 0,016 | 0,0% |
| FC3 | 4096 | | | 1000 | | | | | | | | 4.097.000,00 | 15,629 | 2,96% | 1000 | 0,004 | 0,0% |
| | | | | | | | | | | | | **138.357.544,00** | **527,79** | **1,00** | **15.087.080,00** | **57,55** | **1,00** |

En este mismo año **GoogLeNet**[11], también conocida como **Inception V1**, ganaría la tarea de clasificación con una tasa de error en el top-5 de 6.7%. Su arquitectura se basaría en módulos de convolución en paralelo, conocidos como "*Inception modules*". Estos módulos permiten combinar diferentes tamaños de filtros de convolución en paralelo, lo que permite capturar características a diferentes escalas y resoluciones. Al utilizar módulos de convolución en paralelo, GoogLeNet puede obtiene más características relevantes para la tarea de reconocimiento de imágenes con un número menor de parámetros en el modelo, por lo que este modelo destacó por su eficiencia.

La Figura 12, muestra los detalles más relevantes del módulo con reducción de dimensión. la capa *inception* mostrada en la imagen usa 163.696 parámetros.




| OPERACIÓN | nº filtros | | Salida | Parametros |
|---|---|---|---|---|
| **Rama 1:** conv 1x1 | 64 | | 28x28x64 | (1x1x192+1)x64 = **12.352** |
| **Rama 2:** conv 1x1 -> conv 3x3 | 96 | 128 | 28x28x128 | (1x1x192+1)x96 + (3x3x96+1)x128= **129.248** |
| **Rama 3:** conv 1x1 -> conv 5x5 | 16 | 32 | 28x28x32 | (1x1x192+1)x16 + (5x5x16+1)x32= **15.920** |
| **Rama 4:** pool 3x3 -> conv 1x1 | | 32 | 28x28x32 | (1x1x192+1)x32 = **6.176** |
| **TOTAL:** | | | | **163.696** |

*Figura 12: Capa incepción. Imagen obtenida del trabajo original [11]*

La Tabla 3 muestra el número de parámetros de las diferentes capas, es interesante ver cómo el módulo *Inception* reduce significativamente el número de parámetros a 6.8 millones, lo cual reduce los requerimientos computacionales.

*Tabla 3: Arquitectura **GoogleLeNet** [11]*

| | | | | rama 1 | rama 2 | | rama 3 | | rama 4 | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Tipo** | **Kernel/ Stride** | **Tamaño salida** | **profun-didad** | **conv 1×1** | **conv 1x1** | **conv 3x3** | **conv 1×1** | **conv 5×5** | **pool conv 1x1** | **Parametros** |
| convolution | 7×7/2 | 112×112×64 | 1 | | | | | | | 2.7K |
| max pool | 3×3/2 | 56×56×64 | 0 | | | | | | | |
| convolution | 3×3/1 | 56×56×192 | 2 | 64 | | 192 | | | | 112K |
| max pool | 3×3/2 | 28×28×192 | 0 | | | | | | | |
| inception (3a) | | 28×28×256 | 2 | 64 | 96 | 128 | 16 | 32 | 32 | 159K |
| inception (3b) | | 28×28×480 | 2 | 128 | 128 | 192 | 32 | 96 | 64 | 380K |
| max pool | 3×3/2 | 14×14×480 | 0 | | | | | | | |
| inception (4a) | | 14×14×512 | 2 | 192 | 96 | 208 | 16 | 48 | 64 | 364K |
| inception (4b) | | 14×14×512 | 2 | 160 | 112 | 224 | 24 | 64 | 64 | 437K |
| inception (4c) | | 14×14×512 | 2 | 128 | 128 | 256 | 24 | 64 | 64 | 463K |
| inception (4d) | | 14×14×528 | 2 | 112 | 144 | 288 | 32 | 64 | 64 | 580K |
| inception (4e) | | 14×14×832 | 2 | 256 | 160 | 320 | 32 | 128 | 128 | 840K |
| max pool | 3×3/2 | 7×7×832 | 0 | | | | | | | |
| inception (5a) | | 7×7×832 | 2 | 256 | 160 | 320 | 32 | 128 | 128 | 1072K |
| inception (5b) | | 7×7×1024 | 2 | 384 | 192 | 384 | 48 | 128 | 128 | 1388K |
| avg pool | 7×7/1 | 1×1×1024 | 0 | | | | | | | |
| dropout (40%) | | 1×1×1024 | 0 | | | | | | | |
| linear | | 1×1×1000 | 1 | | | | | | 1000K | 1M |
| softmax | | 1×1×1000 | 0 | | | | | | | |
| | | | | | | | | | **TOTAL** | **6,8 M** |

No obstante, tenía un problema de *vanishing* en los gradientes de los parámetros durante el entrenamiento (*backpropagation*) debido a la profundidad de la red, que hizo que fuera necesario utilizar salidas *softmax* intermedias para la fase de entrenamiento. Para lo cual se utiliza tres términos para la función de pérdida.

En la Figura 13 se muestra la estructura de la red GoogLeNet tal y como aparece publicada en el trabajo original [11], se observan dos salidas *Softmax* intermedias. Se han añadido a la derecha de la imagen las tres componentes de la función de pérdida utilizada durante el entrenamiento.

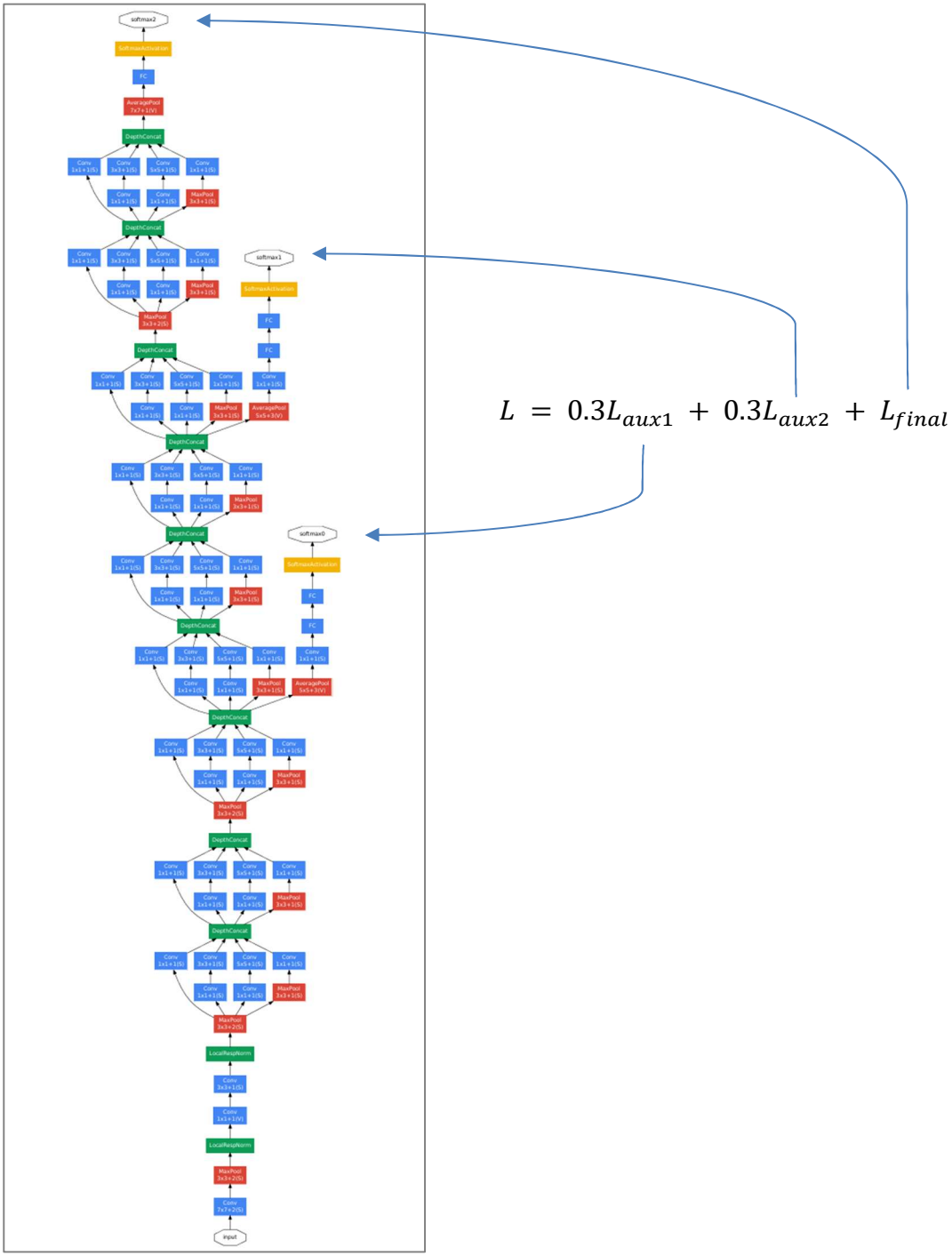



*Figura 13: Estructura de la red GoogLeNet, ilustración obtenida del artículo [11].*

Estos clasificadores auxiliares se eliminan tras el entrenamiento. Esta problemática para transmitir los gradientes en redes profundas durante el proceso de *backpropagation* motiva la estructura de la siguiente red neuronal: ResNet [12]. Hay que comentar también que la degradación en las redes profundas también se debe a la falta de normalización de las capas de activación.

En 2015 se produjo otro gran avance con la introducción de las redes neuronales residuales **ResNet**[12] que alcanzaron una precisión del 3.6%. Esta red permite utilizar muchas más capas ocultas (llegando en el caso de la ResNet hasta 152 capas) sin sufrir de pérdida de rendimiento por imposibilidad de propagar bien el aprendizaje, lo que pasaba en las otras redes es que durante el proceso de retropropagación los gradientes tendían progresivamente a cero.

La red consiste en apilar bloques residuales, como el que se muestra en la Figura 14. De esta manera el aprendizaje se realiza a través de las convoluciones que forman el residuo y los gradientes se propagan directamente a través del cortocircuito, dado que al tratarse de la suma de dos términos el gradiente se distribuye por igual en los dos caminos, de esta manera el gradiente 'sortea' su degradación al pasar por las distintas capas convoluciones cortocircuitadas. En el bloque de la izquierda, la parte dentro del recuadro punteado debe aprender directamente los parámetros que obtenga un mapa de características que reduzca la función de pérdida. En el bloque de la derecha (bloque residual), la parte dentro del recuadro punteado debe aprender los parámetros para que el residuo tienda a cero.

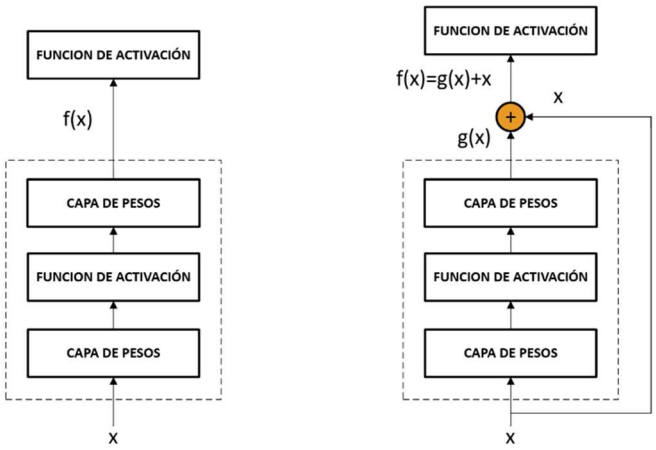



*Figura 14: Bloque normal (izquierda) y con residuo (derecha).*

### 2.1.5 Efecto de la retropropagación en redes neuronales cortocircuitadas.

A continuación, se explica cómo se propaga la salida hacia atrás para actualizar los pesos de las redes

$$f(x_l) = g(x_l) + 1 \qquad (2.8)$$

Obviando la función de activación para simplificar:

$$x_{l+1} = f(x_l) \qquad (2.9)$$

Suponiendo que 'L' es una capa más profunda que 'l', aplicando recursivamente y generalizando obtenemos:

$$x_L = x_l + \sum_{i=l}^{L-1} f(x_i) \qquad (2.8)$$

Para ver el efecto en la *backpropagation* del gradiente se va a considerar la derivada parcial de la función de pérdida **ε** respecto de la entrada $x_l$ al bloque residual.

$$\frac{\partial \varepsilon}{\partial x_l} = \frac{\partial \varepsilon}{\partial x_L}\frac{\partial x_L}{\partial x_l} = \frac{\partial \varepsilon}{\partial x_L}\left(1 + \frac{\partial}{\partial x_l}\sum_{i=l}^{L-1} f(x_i)\right) \qquad (2.9)$$

$$\frac{\partial \varepsilon}{\partial x_l} = \frac{\partial \varepsilon}{\partial x_L} + \frac{\partial \varepsilon}{\partial x_L}\frac{\partial}{\partial x_l}\sum_{i=l}^{L-1} f(x_i) \qquad (2.10)$$

Por lo que el gradiente de una capa menos profunda $\frac{\partial \varepsilon}{\partial x_l}$ se obtiene de la adición de dos términos, un término con $\frac{\partial \varepsilon}{\partial x_L}$ que propaga directamente el gradiente de una capa más profunda sin ser modificado por los pesos de las capas intermedias y otro término $\frac{\partial \varepsilon}{\partial x_L}\frac{\partial}{\partial x_l}\sum_{i=l}^{L-1} f(x_i)$ que se propaga a través de los pesos de las capas entre el cortocircuito. Esta estrategia reduce muchísimo el desvanecimiento del gradiente (*vanishing*), y garantiza que se propague el gradiente, aunque los pesos de las capas intermedias sean muy pequeños. [21]

Dada la clase de función (representada como $F_n$ en la imagen) que una determinada red neuronal junto con sus hiperparámetros puede representar. Esta función de clase con el correcto entrenamiento obtendrá la función de entrenamiento $f_F^*$, de todas la posibles funciones $f \in F$, función de los pesos y bias obtenidos en la fase de entrenamiento. Conforme aumentamos la profundidad de la red esta podrá representar una clase de función más amplia y compleja, pero si esta no está anidada puede alejarse de la función que representa la verdad (*truth function*) representada en la imagen de abajo por $f^*$. En la red no anidada vemos como una red más profunda, en este caso representada como $F_4$ tiene una clase de función más amplia y sin embargo se aleja de la función de verdad $f^*$, obteniendo peores resultados que la $F_3$.

La Figura 15 muestra una representación gráfica de clases de funciones no anidadas (imagen de la izquierda) donde visualmente se aprecia que una clase de función más compleja (representada por un área más amplia) no garantiza acercarse a la función de verdad $f^*$. Este comportamiento no sucede en las funciones de clase anidadas.

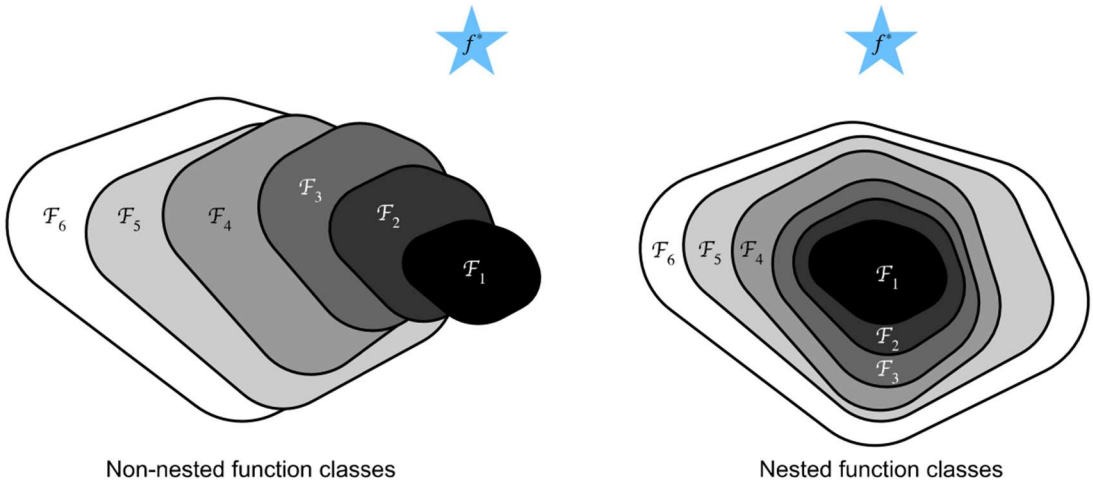



*Figura 15: Representación clases de funciones no anidadas (izquierda) y anidadas (derecha). Representación gráfica obtenida de [20]*

Solo si las funciones de clase más grandes contienen a las más pequeñas se puede garantizar que incrementando el tamaño de la red neural se obtendrá mejor resultado. En el caso de las redes neuronales anidadas podemos entrenar la nueva capa de red neural para que g(x) = 0, y por tanto como si se tratara de una función identidad f(x) = x. De este modo el modelo con más capas será al menos tan efectivo como el original, porque lo contiene:

$$F_1 \subseteq F_2 \subseteq \cdots \subseteq F_6 \qquad (2.11)$$

### 2.1.6 Comparativa redes neuronales convolucionales.

Se muestra a continuación un resumen comparativo de la precisión de los distintos modelos de redes, la complejidad del modelo (número de parámetros) y el número de operaciones en cada propagación completa hacia delante (*single forward pass*).

Alexnet tiene una precisión baja y un tamaño considerable (este tamaño es debido a las capas completamente conectadas del final) sin embargo, computacionalmente es poco exigente. La red VGG tiene mejor precisión que Alexnet, pero usa muchos más parámetros y computacionalmente es muy exigente. GoogLeNet es muy eficiente porque con pocos parámetros y operaciones supera ampliamente a Alexnet. Los modelos ResNet según la cantidad de capas del modelo, están más enfocados al rendimiento o a la precisión. Finalmente, Inception v4 que es básicamente una combinación de Resnet y módulos Inception (introducidos en GoogLeNet) tiene buen equilibrio entre precisión y numero de operaciones.

La Figura 18 muestra en el gráfico de la derecha la precisión de los modelos con la métrica Top1. En el gráfico de la izquierda se observa el tamaño de los modelos expresado como el área del círculo (leyenda en la parte inferior derecha con círculos en gris claro que van de los 5×106 a los 155×106 parámetros) y en eje de abscisas el número de operaciones por cada pasada hacia adelante.[22]

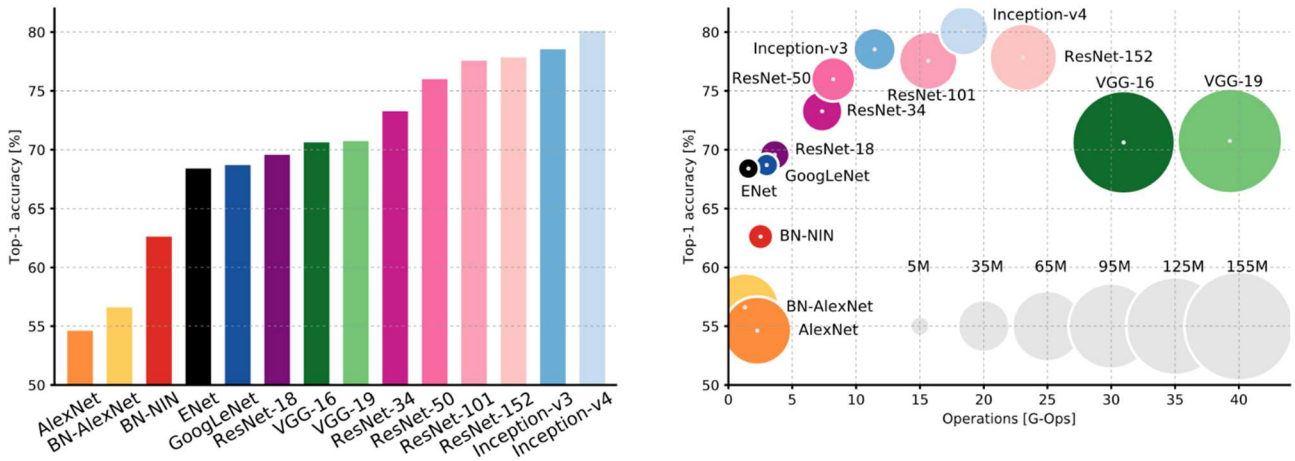



*Figura 16: Comparativa redes neuronales modernas. Ilustración obtenida de [22].*

## 2.2. Detección de objetos y métricas

### 2.2.1 Introducción y conceptos generales

La detección de objetos es una tarea fundamental en visión artificial que implica identificar y localizar objetos dentro de una imagen o video. En este campo, se busca desarrollar algoritmos que puedan detectar objetos de interés, como personas, vehículos, objetos cotidianos, entre otros, y localizarlos en el espacio de la imagen.

Es por tanto un proceso que combina la clasificación y la localización de objetos en una imagen. Esto significa que los detectores de objetos pueden:

- Clasificar objetos: identificar y categorizar objetos dentro de las clases para las que fueron entrenados. Esto produce vector de salida con la precisión asociada a cada clase.

- Localizar objetos: automáticamente, se genera un área que delimita el objeto clasificado en la imagen. Esto se logra mediante un rectángulo delimitador llamado *bounding box*.

### 2.2.2 Bounding Box y Truth Bounding:

En la detección de objetos, se utiliza el concepto de *bounding box* (caja delimitadora) para representar la región de interés que contiene el objeto. Un *bounding box* es un rectángulo que rodea el objeto y se define por sus coordenadas (x, y, w, h), donde (x, y) es el centro del objeto y w y h son la anchura y altura del objeto, respectivamente.

En una detección puede haber varios *bounding box* para el mismo objeto, pero también puede haber solapamiento de *bounding boxes* de diferentes objetos incluso de la misma clase. Se explicará más adelante técnicas para reducir este número detecciones redundantes del mismo objeto, como el método de supresión no máxima.

El *truth bounding* (delimitador de verdad) se refiere al *bounding box* que se utiliza como etiqueta de verdad para entrenar y evaluar los algoritmos de detección de objetos. El *truth bounding* es fundamental para evaluar la precisión de los algoritmos de detección de objetos.

### 2.2.3 Enfoques para la Detección de Objetos

En la detección de objetos, se han desarrollado varios enfoques utilizando redes neuronales convolucionales (CNN). Estos enfoques se pueden clasificar en dos categorías principales:

- Enfoque de dos etapas: En este enfoque, se utiliza una red neuronal convolucional para detectar regiones de interés (RoI) en la imagen y luego se aplica una segunda red neuronal convolucional para clasificar y refinar la detección de objetos en cada región de interés. Algunos ejemplos de algoritmos que utilizan este enfoque son Faster R-CNN [6], R-FCN [13] y Mask R-CNN [14].

- Enfoque de una etapa: En este enfoque, se utiliza una sola red neuronal convolucional para detectar objetos y clasificarlos en una sola etapa. Algunos ejemplos de algoritmos que utilizan este enfoque son YOLO (*You Only Look Once*), SSD (*Single Shot Detector*) y RetinaNet [15].

### 2.2.4 Intersecction over Union (IoU)

Es una métrica utilizada en visión por computadora y procesamiento de imágenes para evaluar la precisión de la detección de objetos en imágenes. Se utiliza para calcular la similitud entre dos *bounding boxes* (cajas delimitadoras) que rodean un objeto en una imagen.

La IoU se define como la relación entre el área de intersección entre dos *bounding boxes* y el área de unión entre ellas.

Matemáticamente, se expresar como:

$$IoU = \frac{(Área\ de\ intersección)}{(Área\ de\ unión)} \tag{2.12}$$

donde:

Área de intersección: es el área común entre las dos *bounding boxes*.
Área de unión: es el área total de las dos *bounding boxes*, incluyendo la intersección.

Supongamos que tenemos una imagen con un objeto que queremos detectar, y que la etiqueta de verdad es una *bounding box* con coordenadas (x1, y1, x2, y2). El modelo predice una *bounding box* con coordenadas (x1', y1', x2', y2').

Para calcular el IoU, necesitamos calcular las áreas de intersección y unión:

$$Intersección = \max(0, \min(x2, x2') - \max(x1, x1')) * max(0, min(y2, y2') - max(y1, y1'))$$

$$Unión = (x2 - x1) * (y2 - y1) + (x2' - x1') * (y2' - y1') - Intersección$$

$$IoU = \frac{Intersección}{Unión}$$

Interpretación de los valores de IoU varía entre 0 y 1, donde:

IoU = 0 indica que las *bounding boxes* no se intersectan.
IoU = 1 indica que las *bounding boxes* son idénticas.
IoU > 0.5 indica que las *bounding boxes* se intersectan significativamente.

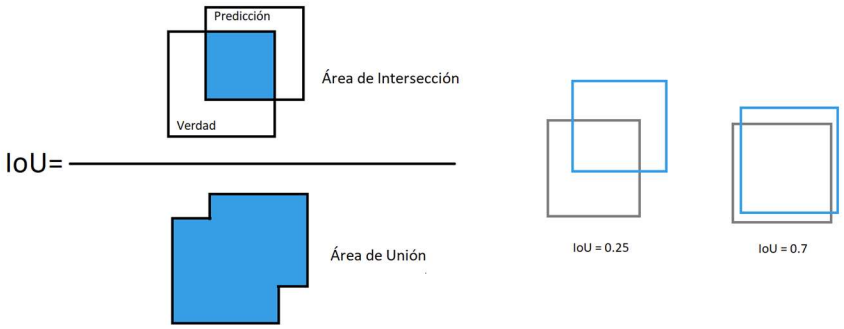



*Figura 17: Significado gráfico del IoU.*

### 2.2.5 Supresión no máxima (*Non Maximum Supresion*)

La supresión no máxima (en adelante NMS) es una técnica que se utiliza después del paso de propuesta de regiones para eliminar detecciones duplicadas y seleccionar los objetos detectados más relevantes.

Funciona comparando las puntuaciones de confianza de las *bounding boxes* propuestas y eliminando las que se superponen significativamente con una *bounding box* con una puntuación de confianza más alta.

El proceso de NMS sería el siguiente:

1. Ordenar las *bounding boxes*: Ordenar las *bounding boxes* en orden descendente de sus puntuaciones de confianza.

2. Seleccionar la *bounding box* con la puntuación de confianza más alta: Seleccionar la *bounding box* con la puntuación de confianza más alta y guardarla como una detección.

3. Eliminar todas las *bounding boxes* que tengan una superposición significativa con la *bounding box* seleccionada: Calcular la intersección sobre la unión (IoU) entre la *bounding box* seleccionada y todas las *bounding boxes* restantes. Si el valor de IoU supera un umbral predefinido, eliminar la *bounding box* con la puntuación de confianza más baja.

La NMS reduce significativamente el número de falsos positivos en los resultados de detección de objetos. Los falsos positivos ocurren cuando se genera una *bounding box* para un área de la imagen que no contiene un objeto. La NMS elimina estos falsos positivos seleccionando solo las *bounding boxes* más relevantes que corresponden a los objetos detectados y reduce las detecciones redundantes. Hay variantes con ponderación del peso de los scores, pero la idea principal es la misma.

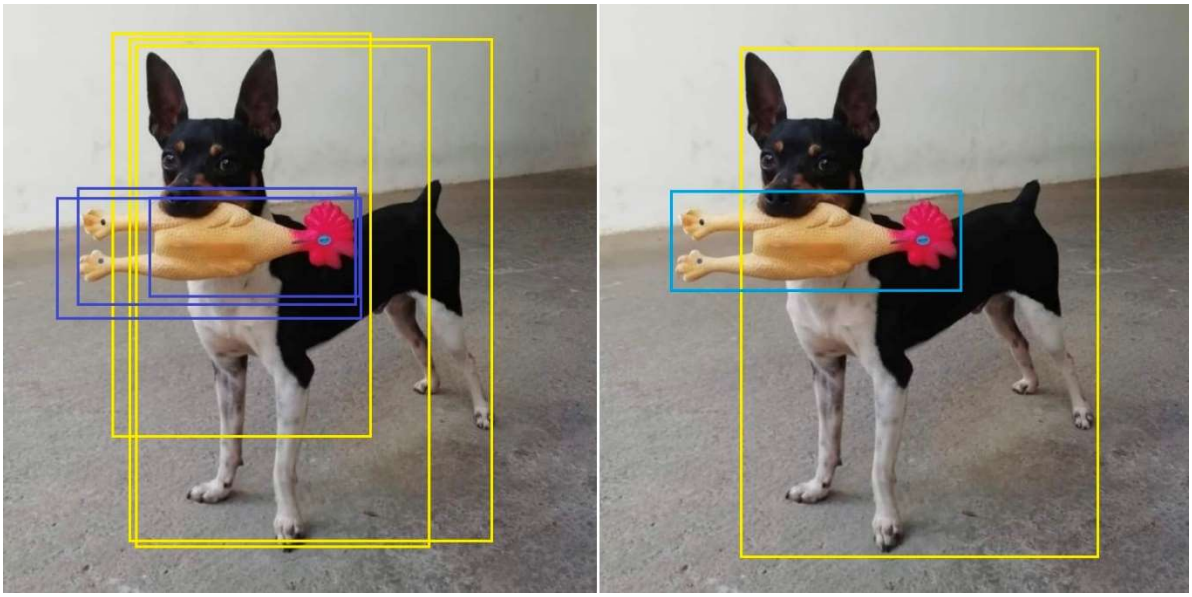


*Figura 18: Ejemplo gráfico de supresión no máxima.*

### 2.2.6 Métricas

Se introducen en este apartado la matriz de confusión y curva *precision-recall*:

**Matriz de confusión:**

La matriz de confusión es una herramienta fundamental en el contexto de detección y segmentación en visión artificial, ya que permite evaluar el rendimiento de un modelo a la hora de clasificar y detectar de objetos en imágenes.

Es una tabla que muestra la relación entre los resultados predichos por un modelo y los resultados reales en una tarea de clasificación o detección. La matriz se compone de cuatro elementos:

- Verdaderos Positivos (TP): número de objetos que se han detectado correctamente como pertenecientes a la clase de interés.
- Falsos Positivos (FP): número de objetos que se han detectado incorrectamente como pertenecientes a la clase de interés, cuando en realidad no lo son.
- Verdaderos Negativos (TN): número de objetos que no se han detectado como pertenecientes a la clase de interés, y que efectivamente no lo son.
- Falsos Negativos (FN): número de objetos que no se han detectado como pertenecientes a la clase de interés, cuando en realidad sí lo son.

Supongamos que estamos entrenando un modelo para detectar objetos en una imagen, y que la clase de interés como en nuestro caso "flejes". La matriz de confusión para este modelo podría ser la siguiente:

*Tabla 4: Ejemplo matriz de confusión de una sola clase.*

| | Predicción Positiva | Predicción Negativa |
|---|---|---|
| Clase Real Positiva | 80 (TP) | 20 (FN) |
| Clase Real Negativa | 30 (FP) | - |

En este ejemplo, el modelo ha detectado correctamente 80 flejes (TP), pero también ha detectado incorrectamente 30 objetos como flejes (FP). Por otro lado, ha fallado en detectar 20 flejes que están presentes en la imagen (FN). En el caso de los verdaderos negativos (TN) no haremos referencias, porque son todos los fondos de las imágenes.

En detección de objetos, el índice IoU (*Intersection over Union*) es una métrica comúnmente utilizada para determinar si una detección es correcta o no. Para lo cual, se establece un umbral de IoU. Por regla general, si el IoU es mayor o igual a 0.5, se considera que la detección es correcta. Si el IoU es menor que 0.5, se considera que la detección es incorrecta.

En segmentación, la determinación se hace en función de los píxeles de *ground truth*, de tal modo que para cada detección se podrán tener píxeles que son TP, píxeles que son FP y píxeles que son FN.

A partir de la matriz de confusión, se pueden calcular varias métricas que permiten evaluar el rendimiento del modelo:

- **Precisión *(Precision)*:** número de verdaderos positivos dividido entre el número total de predicciones positivas (TP + FP).

$$Precision = \frac{TP}{TP+FP} \quad (2.13)$$

  En el ejemplo, la precisión sería 80 / (80 + 30) = 0.73.

- **Sensibilidad *(Recall)*:** número de verdaderos positivos dividido entre el número total de objetos reales positivos (TP + FN).

$$Recall = \frac{TP}{TP+FN} \quad (2.14)$$

  En el ejemplo, el recall sería 80 / (80 + 20) = 0.80.

**Curva *Precision-Recall*:**

Es un diagrama de líneas donde se coloca la Precisión en el eje de ordenadas y el *Recall* en el eje de abscisas, se construye calculando la precisión y el *Recall* para diferentes umbrales de clasificación. Por ejemplo, si estamos evaluando un detector de objetos, podemos variar el umbral de confianza del detector y calcular la precisión y el *recall* en cada punto. La curva precisión-*recall* se obtiene uniendo los puntos (*recall*, precisión) correspondientes a cada umbral de clasificación.

La interpretación de los resultados de la curva *precisión-recall* depende del contexto y los objetivos del clasificador o detector. En general, un clasificador o detector con una curva *precision-recall* más alta es preferible a uno con una curva más baja, ya que indica una mejor relación entre la precisión y el *recall*.

El área bajo esta curva constituye la base para calcular la metrica **mAP (*Mean Average Precition*)**, fundamental para determinar el rendimiento de un modelo de detección.

En la Figura 19 se muestra la curva *Precion-Recall* obtenida con YOLO v8 para la clase radial, cirunferencial y la media de las dos clases. El área bajo la curva constituye el mAP para un criterio de IoU >0.5

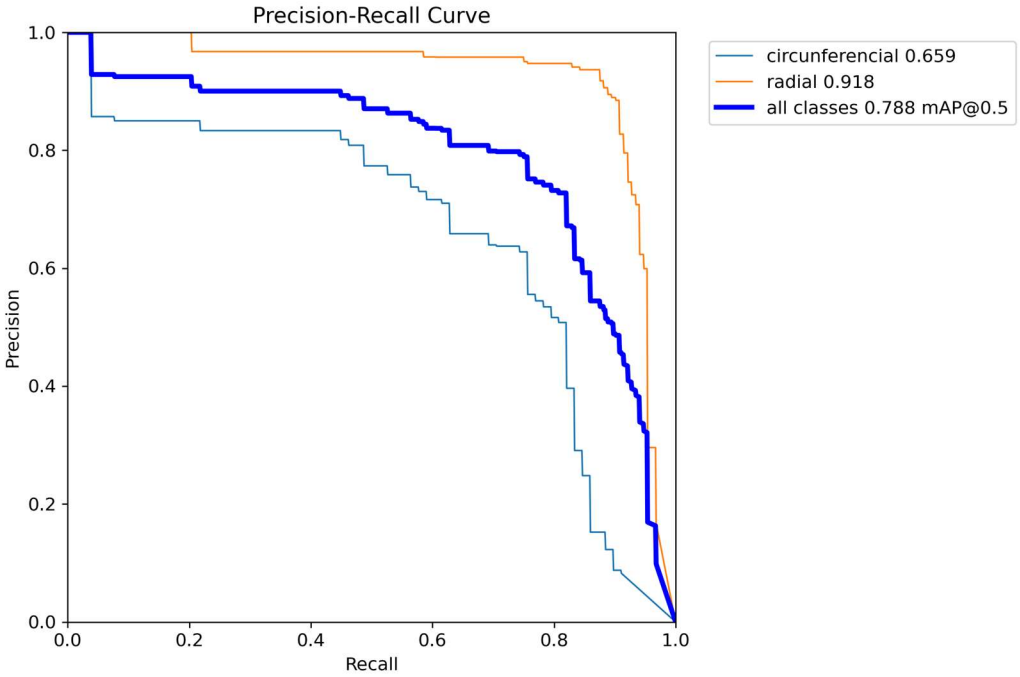



*Figura 19: Curva Precion-Recall con criterio de IoU >0.5. Obtenida de los resultados de YOLO para este proyecto.*

## 2.3. YOLO *(You Only Look Once: Unified, Real-Time Object Detection)*

La primera versión de YOLO [23] se publica en 2016 e introduce un enfoque innovador para detectar objetos en imágenes. Divide la imagen en una cuadrícula de celdas y mediante una red neuronal convolucional predice la probabilidad de que cada celda contenga un objeto, la posición, tamaño y la predicción de clase, todo esto en un solo paso hacia delante (*single forward pass*) de la CNN. Replantea la detección de objetos como un problema de regresión, directamente desde los píxeles de la imagen hasta las coordenadas del cuadro delimitador y las probabilidades de clase.

Otros métodos utilizados hasta ese momento para detectar objetos consistían en usar ventanas deslizantes seguidas de un clasificador o métodos más avanzados que dividían la operación en dos pasos: un primer paso para detectar posibles regiones con objetos (*regions proposals*) y un segundo paso que corre un clasificador sobre la región propuesta. Por tanto, debido a estos dos pasos eran bastante más lentos que YOLO. Dentro de este grupo se encuentra la familia conocida como R-CNN [4].

A diferencia de las técnicas basadas en '*regions proposals*', YOLO ve la imagen completa durante el entrenamiento por lo que codifica mejor la información de contexto de las distintas clases así como su apariencia, esto hace que tenga menos errores de fondo comparado con Fast R-CNN [5]. YOLO aprende mejor la representación generalizable de los objetos, entrenado con imágenes naturales (de la vida real) es capaz de detectar objetos de distinta naturaleza, por ejemplo, en obras de arte (pinturas), y lo hace mejor que lo hacían los métodos de su época.

Como debilidad a pesar de poder identificar los objetos tiene peor precisión a la hora de localizarlos, especialmente los objetos pequeños.

### 2.3.1 Descripción del método

La imagen de entrada se divide en una cuadrícula de S×S celdas, en cada celda se predicen B cajas delimitadoras (*bounding boxes*) todas de la misma clase, así como la predicción de clase con la confianza de las C diferentes clases (predicción común a las B cajas delimitadoras de la misma celda). Cada predicción de caja delimitadora consiste en 5 valores: [*Pc*, *bx*, *by*, *bh*, *bw*], donde *Pc* es la confianza de que caja contenga un objeto (da igual la clase), *bx* y *by* son las coordenadas del centro de la caja delimitadora referidas a las coordenadas relativas de la celda, *bh* y *bw* son el alto y ancho de la caja relativas al tamaño de la imagen completa. Todas estas salidas están contenidas por tanto entre 0 y 1. La salida de YOLO será un tensor de tamaño [S,S,(B×5+C)]

En el modelo original se utilizaba un tamaño de cuadricula de 7×7 celdas (S=7), dos cajas delimitadoras por celda (B = 2) y como se utilizaba el dataset PASCAL VOC2007 se hacía una predicción sobre 20 clases (C = 20). Por lo que la salida de la CNN del documento original era un tensor de tamaño [7,7,30]. La precisión media obtenida en su origen fue mAP = 0,634.

En YOLO V1 el *dataset* utilizado es PASCAL VOC 2007 [24] el cual tiene 20 categorías de objetos y la forma del calcular el mAP difiere del usado en el dataset COCO (*Common Objects in Context*) de Microsoft [25] que es más complejo y tiene 80 clases. Por lo que el rendimiento del modelo no puede ser comparado con las métricas de la V3 en adelante.

La arquitectura de YOLO está compuesta por 24 capas convolucionales seguidas de 2 capas completamente conectadas, Figura 20.

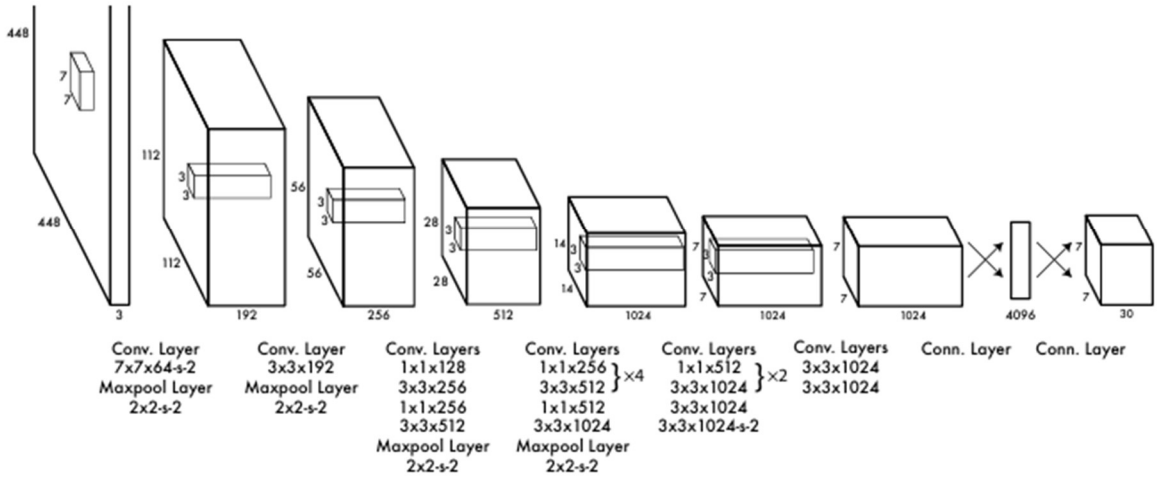



*Figura 20: Arquitectura de YOLO. Imagen obtenida del artículo original [23]*

La Tabla 5 muestra el número de parámetros y de activaciones de la red original YOLO[23], también se estima el tamaño del modelo y de las activaciones, suponiendo que se usa un float32. Las capas convolucionales iniciales extraen las características de la imagen y las dos capas completamente conectadas a la salida predicen las coordenadas y probabilidad de la salida: Tensor 7×7×30

*Tabla 5: Arquitectura YOLOV1*

| | | | | | | | | | | | | | | | float32 data type | | | | float32 data type | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **YOLO ORIGINAL V1** | | | | | | | | | | | | | | | MB | | | | MB | |
| | **CAPA** | **ENTRADA** | | | **SALIDA** | | | **KERNEL** | | **STRIDE** | **PAD** | **Nº FILT.** | **PARAMETROS** | | **MEMORIA PARAMETROS** | **%** | **ACTIVACIONES** | | **MEMORIA ACTIVACIONES** | **%** |
| | CONV | 448 | 448 | 3 | 224 | 224 | 64 | 7 | 7 | 2 | 3 | 64 | 9.472,00 | | 0,036 | 0,00% | 3.211.264,00 | | 12,250 | 20,3% |
| | MAX POOL | 224 | 224 | 64 | 112 | 112 | 64 | 2 | 2 | 2 | 0 | | - | | - | 0,00% | 802.816,00 | | 3,063 | 5,1% |
| | CONV | 112 | 112 | 64 | 112 | 112 | 192 | 3 | 3 | 2 | 0 | 192 | 110.784,00 | | 0,423 | 0,04% | 2.408.448,00 | | 9,188 | 15,2% |
| | MAX POOL | 112 | 112 | 192 | 56 | 56 | 192 | 2 | 2 | 2 | 0 | | - | | - | 0,00% | 602.112,00 | | 2,297 | 3,8% |
| | CONV | 56 | 56 | 192 | 56 | 56 | 128 | 1 | 1 | 1 | 0 | 128 | 24.704,00 | | 0,094 | 0,01% | 401.408,00 | | 1,531 | 2,5% |
| | CONV | 56 | 56 | 128 | 56 | 56 | 256 | 3 | 3 | 1 | 1 | 256 | 295.168,00 | | 1,126 | 0,10% | 802.816,00 | | 3,063 | 5,1% |
| | CONV | 56 | 56 | 256 | 56 | 56 | 256 | 1 | 1 | 1 | 0 | 256 | 65.792,00 | | 0,251 | 0,02% | 802.816,00 | | 3,063 | 5,1% |
| | CONV | 56 | 56 | 256 | 56 | 56 | 512 | 3 | 3 | 1 | 1 | 512 | 1.180.160,00 | | 4,502 | 0,42% | 1.605.632,00 | | 6,125 | 10,2% |
| | MAX POOL | 56 | 56 | 512 | 28 | 28 | 512 | 2 | 2 | 2 | 0 | | - | | - | 0,00% | 401.408,00 | | 1,531 | 2,5% |
| x4 | CONV | 28 | 28 | 512 | 28 | 28 | 256 | 1 | 1 | 1 | 0 | 256 | 525.312,00 | | 2,004 | 0,19% | 802.816,00 | | 3,063 | 5,1% |
| | CONV | 28 | 28 | 256 | 28 | 28 | 512 | 3 | 3 | 1 | 1 | 512 | 4.720.640,00 | | 18,008 | 1,67% | 1.605.632,00 | | 6,125 | 10,2% |
| | CONV | 28 | 28 | 512 | 28 | 28 | 512 | 1 | 1 | 1 | 0 | 512 | 262.656,00 | | 1,002 | 0,09% | 401.408,00 | | 1,531 | 2,5% |
| | CONV | 28 | 28 | 512 | 28 | 28 | 1024 | 3 | 3 | 1 | 1 | 1024 | 4.719.616,00 | | 18,004 | 1,67% | 802.816,00 | | 3,063 | 5,1% |
| | MAX POOL | 28 | 28 | 1024 | 14 | 14 | 1024 | 2 | 2 | 2 | 0 | | | | - | 0,00% | 200.704,00 | | 0,766 | 1,3% |
| x2 | CONV | 14 | 14 | 1024 | 14 | 14 | 512 | 1 | 1 | 1 | 0 | 512 | 1.049.600,00 | | 4,004 | 0,37% | 200.704,00 | | 0,766 | 1,3% |
| | CONV | 14 | 14 | 512 | 14 | 14 | 1024 | 3 | 3 | 1 | 1 | 1024 | 9.439.232,00 | | 36,008 | 3,34% | 401.408,00 | | 1,531 | 2,5% |
| | CONV | 14 | 14 | 1024 | 14 | 14 | 1024 | 3 | 3 | 1 | 1 | 1024 | 9.438.208,00 | | 36,004 | 3,34% | 200.704,00 | | 0,766 | 1,3% |
| | CONV | 14 | 14 | 1024 | 7 | 7 | 1024 | 3 | 3 | 2 | 1 | 1024 | 9.438.208,00 | | 36,004 | 3,34% | 50.176,00 | | 0,191 | 0,3% |
| | CONV | 7 | 7 | 1024 | 7 | 7 | 1024 | 3 | 3 | 1 | 1 | 1024 | 9.438.208,00 | | 36,004 | 3,34% | 50.176,00 | | 0,191 | 0,3% |
| | CONV | 7 | 7 | 1024 | 7 | 7 | 1024 | 3 | 3 | 1 | 1 | 1024 | 9.438.208,00 | | 36,004 | 3,34% | 50.176,00 | | 0,191 | 0,3% |
| | FC | 50176 | | | 4096 | | | | | | | 4096 | 205.524.992,00 | | 784,016 | 72,76% | 4096 | | 0,016 | 0,0% |
| | FC | 4096 | | | 4096 | | | | | | | 1470 | 16.781.312,00 | | 64,016 | 5,94% | 4096 | | 0,016 | 0,0% |
| | | | | | Tensor 7 x 7 x 30 | | | | | | | | **282.462.272,00** | | **1.077,51** | **100%** | **15.813.632,00** | | **60,32** | **100%** |

### 2.3.2 Entrenamiento de la red

El entrenamiento del primer modelo original se realiza en dos pasos. Primero se entrenan las primeras 20 capas convolucionales con el *dataset* the ImageNet (1000 clases). Para lo cual se añaden un *average-pooling* seguido de una capa completamente conectada con 1000 salidas a las primeras 20 capas convolucionales. Para la función de pérdida se utilizó error cuadrático medio (Mean Squared Error, MSE) tanto para la regresión (coordenadas de las cajas) como para la clasificación. Entrenan con el hardware del momento durante una semana, hasta obtener una presión en top-5 del 88% en los datos de validación de ImageNet 2012. Esto permite entrenar el modelo para captar muchas características de la imagen, pero sin centrarse aun en la localización, permitiendo usar un *dataset* de clasificación (normalmente más extenso que los de detección) para pre-entrenar un modelo que será finalmente de detección.

En una segunda fase se entrenará al modelo para el problema de detección, para lo cual se añaden 4 capas convolucionales más y 2 completamente conectadas a las 20 capas convolucionales. El *average-pooling* y la FC del primer entrenamiento se desechan. Estas capas nuevas se inicializan con pesos aleatorios. Como la detección requiere más resolución, la dimensión de las imágenes de entrada se aumenta de 224×244 a 448×448.

Como función de activación para la última capa completamente conectada se utiliza una función de activación lineal (realmente no es una capa de activación pues no introduce no linealidad) se utiliza esta función de activación porque la salida de YOLO es la propia de un problema de regresión. Para el resto de capas intermedias se utiliza la función Leaky-ReLU.

Para evitar el sobreajuste se emplea *dropout* de 0.5 después de la primera capa completamente conectada. Para el *data-augmentation* usa un escalado aleatorio y traslaciones de la imagen original de hasta el 20%. Se hacen también ajuste aleatorios de la exposición y saturación de la imagen. El aumento de datos (*data-augmentation*) consiste en la generación de nuevas imágenes a partir de las existentes sobre las que se realizan diferentes operaciones geométricas y radiométricas.

Se muestra a continuación la función de pérdida compuesta por varios términos: pérdida de localización, pérdida de confianza y pérdida de clasificación:

Término de la función de pérdida para la localización:

$$L = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2]$$
$$+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[ \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2 \right]$$

Término de la función de pérdida para la confianza para objetos presentes o ausentes:

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left(C_i - \hat{C}_i\right)^2$$
$$+ \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} \left(C_i - \hat{C}_i\right)^2$$

Término de la función de pérdida para la clasificación:

$$+\sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in classes} \left(p_{i(c)} - \hat{p}_{i(c)}\right)^2 \qquad (2.15)$$

Para la función de pérdida se utiliza la suma del error al cuadrado. Para lo cual se realizan algunos ajustes, como dar más peso al error de localización multiplicando por una cte $\lambda_{coord} = 5$ tanto para la localización como para la predicción del tamaño. Solo se aplica sobre las cajas delimitadoras que contengan un objeto y esta sea la responsable de detectarlo, es decir solo una caja delimitadora por objeto, aquella que tenga mayor IOU con el *Ground Truth*.

La suma al cuadrado del error en cajas delimitadoras pequeñas produce valores más pequeños que en cajas grandes, y por tanto menos peso a la hora de entrenar el modelo para detectar objetos pequeños (genera menor gradiente). Para compensar (solo en parte) este problema se aplica la raíz cuadrada del valor de ancho y alto antes de calcular el error al cuadrado.

Se reduce el peso de pérdida para las cajas delimitadoras que no contengan objetos, dado que la mayoría no van a tener objetos y se pretende reducir su peso sobre la función de pérdida. Este lleva a usar una cte $\lambda_{noobj} = 0.5$.

En cuanto al termino de clasificación no se usa entropía cruzada para la parte de clasificación, se usa el sumatorio del cuadrado del error. Solo penaliza el error de clasificación si un objeto está presente en la celda.

### 2.3.3 Inferencia

Durante la inferencia, la red neuronal se utiliza para detectar objetos en imágenes nuevas. La salida de la red neuronal se procesa para obtener la posición y la clase del objeto, y se utiliza un umbral de confianza para determinar si el objeto es del tipo VP o FP.

Cuando un objeto puede ser localizado en varias celdas se aplica la técnica de Non-Maximum Suppression (NMS) para eliminar las detecciones duplicadas. Esta técnica se utiliza para eliminar cajas delimitadoras redundantes y quedarse solo con la más precisa.

### 2.3.4 Limitaciones

YOLO tiene importantes restricciones espaciales, pues cada celda solo contiene 2 cajas delimitadoras y solo puede existir una clase dentro de la celda. Este modelo tiene por tanto dificultades para detectar objetos pequeños, sobre todo si aparecen en grupo.

Tiene dificultades para generalizar objetos que tienen distinta forma o aspecto que en los datos de entrenamiento. Además, la función de pérdida trata igual al error en cajas delimitadoras grandes que pequeñas, pero el error en cajas pequeñas tiene mucho mayor efecto sobre el IOU, por lo que la principal fuente de error de este método es de localización.

### 2.3.5 YOLO V2 [26]

Se introducen ciertas mejoras, respecto de la versión inicial, enfocadas principalmente a mejorar la sensibilidad (*recall*) y la localización, manteniendo la precisión en la clasificación del modelo anterior:

a) Normalización por lotes (*Batch Normalization*): mejora la convergencia y ayuda a regularizar el modelo. Se añade en todas las capas convolucionales obteniéndose una mejora de 2% en el mAP. Mejora la capacidad del modelo para generalizar sin necesidad de utilizar *dropout* para evitar el sobreajuste. Reduce la dependencia de la iniciación de las constantes y *bias* del modelo.

b) Clasificador de alta resolución (*High-Resolution Classifier*): en la versión original de YOLO el pre-entrenamiento en ImageNet se realiza con una resolución de 224×224, en YOLO V2 se pre-entrena directamente con resolución 448×448. Luego se entrenan las ultimas capas de detección con esa misma resolución, esto permite al modelo captar las características de la imagen desde el principio a la máxima resolución. Según el trabajo origina, esto supone una mejora del 4% en el mAP.

c) Convolución con cajas de ajuste (*Convolutional with Anchor Boxes*): el modelo se modifica para trabajar con *anchor boxes*. Esto permite al modelo trabajar con *offsets* en lugar de valores absolutos de coordenadas, lo cual facilita el aprendizaje. Aunque no produce una mejora en la métrica mAP, sí que mejora el Recall del 81% a 88%.

d) Agrupamiento de dimensiones (*Dimension Clusters*): Se agrupan las dimensiones de las cajas delimitadoras utilizando agrupamiento mediante *K-means*. Este enfoque ayuda a diseñar anclas más adecuadas a los tamaños reales de los objetos presentes en los datos de entrenamiento, en lugar hacer configuraciones hechas a mano. Como resultado, mejora la eficiencia y precisión del modelo. Para que el tamaño de la caja delimitadora no influya en la métrica utilizada para *K-means* no se utiliza la distancia euclídea, sino que será función de IOU:

$$d(caja, centroide) = 1 - IOU(caja, centroide) \qquad (2.16)$$

e) Predicción directa de ubicación (*Direct Location Prediction*): Predecir directamente la localización de la cajas produce problemas de instabilidad, por lo que el enfoque utilizado es predecir las coordenadas de localización relativas a la rejilla (*grid*), tal y como se hace en la versión original de YOLO. Se utiliza una sigmoide σ() para forzar que la localización caiga siempre dentro de la cuadricula, restringiendo los valores entre 0 y 1.

   En la Figura 21 se muestra una predicción de caja delimitadora. Las coordenadas del centro son obtenidas con los valores predichos tx y ty a través de la función sigmoide σ() y desplazados por la localización de la celda cx y cy. Para el ancho y alto se usan los valores predimensionados pw y ph de las cajas de anclaje escalados por la exponente del valor predecido tw y th.

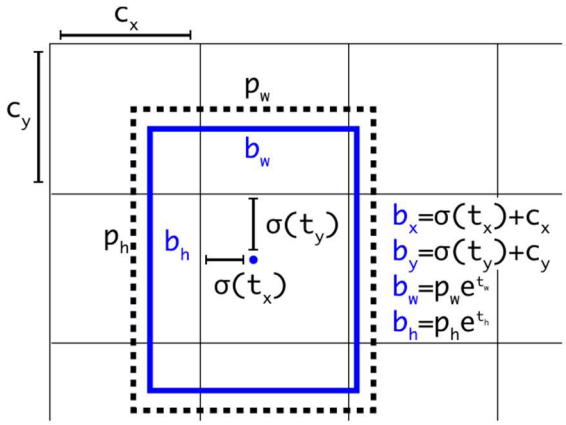



*Figura 21: Predicción con caja delimitadora. Representación gráfica obtenida de [26]*

La combinación *K-means* con valor k=5 para el pre-dimiensionado de 5 cajas de anclajes por celda, junto con el método para localizar el centroide mejora un 5% el mAP.

f) Características detalladas (*Fine-Grained Features*): comparado con la primera versión de YOLO, esta versión elimina una capa de *pooling*, obteniendo un mapa de característica de 13×13 para una imagen de entrada de 416×416. Además, usa un atajo para concatenar una capa intermedia de 26 ×26×512, reordenada a ×13×2048, con una capa más profunda y con la resolución final de 13×13×1024. Esto supone recordar información espacial añadiendo nuevos canales. Se obtiene una mejora en el rendimiento de 1% en mAP.

g) Multi-Scale Training (*Multi-Scale Training*): dado que el modelo usa solo capas convolucionales y pool, puede redimensionar al vuelo. El modelo se entrena utilizando imágenes de diferentes tamaños cada10 batches, de forma aleatoria y con un factor de 32, siendo la resolución mínima 320×320 y la máxima 608×608.

En la Figura 22 se observa la contribución de cada una de las mejoras sobre el rendimiento del modelo. A pesar de que las cajas de anclaje (anchor boxes) apenas tienen efecto en el mAP mejoran el Recall. El uso de predimensionado de las cajas de anclaje mediante *K-means* usando los datos de entrenamiento y el método de predicción de la localización mejoran significativamente el mAP.

| | YOLO | | | | | | | | YOLOv2 |
|---|---|---|---|---|---|---|---|---|---|
| batch norm? | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| hi-res classifier? | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| convolutional? | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| anchor boxes? | | | | ✓ | ✓ | | | | |
| new network? | | | | | ✓ | ✓ | ✓ | ✓ | ✓ |
| dimension priors? | | | | | | ✓ | ✓ | ✓ | ✓ |
| location prediction? | | | | | | ✓ | ✓ | ✓ | ✓ |
| passthrough? | | | | | | | ✓ | ✓ | ✓ |
| multi-scale? | | | | | | | | ✓ | ✓ |
| hi-res detector? | | | | | | | | | ✓ |
| VOC2007 mAP | 63.4 | 65.8 | 69.5 | 69.2 | 69.6 | 74.4 | 75.4 | 76.8 | **78.6** |

*Figura 22: Contribución de cada una de las mejoras sobre el rendimiento de YOLO V2. Obtenido de [26]*

La arquitectura de YOLO V2 se basa en una columna vertebral (*backbone*) llamada Darknet19, esta usa convoluciones 1×1 entre convoluciones 3×3 para reducir el número de parámetros. Todas las capas usan normalización por lotes (*batch normalization*).

*Tabla 6: Arquitectura YOLO V2*

| | | | | | | | | | | | | | | float32 data type | | | | float32 data type | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **YOLO V2** | | | | | | | | | | | | | MB | | | | MB | |
| | **CAPA** | **ENTRADA** | | | **SALIDA** | | | **KERNEL** | | **STRIDE** | **PAD** | **Nº FILT.** | **PARAMETROS** | **MEMORIA PARAMETROS** | **%** | **ACTIVACIONES** | **MEMORIA ACTIVACIONES** | **%** |
| Darknet 19 BACKBONE | CONV1 | 416 | 416 | 3 | 416 | 416 | 32 | 3 | 3 | 1 | 1 | 32 | 896,00 | 0,003 | 0,00% | 5.537.792,00 | 21,125 | 29,5% |
| | MAX POOL1 | 416 | 416 | 32 | 208 | 208 | 32 | 2 | 2 | 2 | 0 | | - | - | 0,00% | 1.384.448,00 | 5,281 | 7,4% |
| | CONV2 | 208 | 208 | 32 | 208 | 208 | 64 | 3 | 3 | 1 | 1 | 64 | 18.496,00 | 0,071 | 0,03% | 2.768.896,00 | 10,563 | 14,7% |
| | MAX POOL2 | 208 | 208 | 64 | 104 | 104 | 64 | 2 | 2 | 2 | 0 | | - | - | 0,00% | 692.224,00 | 2,641 | 3,7% |
| | CONV3 | 104 | 104 | 64 | 104 | 104 | 128 | 3 | 3 | 1 | 1 | 128 | 73.856,00 | 0,282 | 0,11% | 1.384.448,00 | 5,281 | 7,4% |
| | CONV4 | 104 | 104 | 128 | 104 | 104 | 64 | 1 | 1 | 1 | 0 | 64 | 8.256,00 | 0,031 | 0,01% | 692.224,00 | 2,641 | 3,7% |
| | CONV5 | 104 | 104 | 64 | 104 | 104 | 128 | 3 | 3 | 1 | 1 | 128 | 73.856,00 | 0,282 | 0,11% | 1.384.448,00 | 5,281 | 7,4% |
| | MAX POOL | 104 | 104 | 128 | 52 | 52 | 128 | 2 | 2 | 2 | 0 | | - | - | 0,00% | 346.112,00 | 1,320 | 1,8% |
| | CONV6 | 52 | 52 | 128 | 52 | 52 | 256 | 3 | 3 | 1 | 1 | 256 | 295.168,00 | 1,126 | 0,44% | 692.224,00 | 2,641 | 3,7% |
| | CONV7 | 52 | 52 | 256 | 52 | 52 | 128 | 1 | 1 | 1 | 0 | 128 | 32.896,00 | 0,125 | 0,05% | 346.112,00 | 1,320 | 1,8% |
| | CONV8 | 52 | 52 | 128 | 52 | 52 | 256 | 3 | 3 | 1 | 1 | 256 | 295.168,00 | 1,126 | 0,44% | 692.224,00 | 2,641 | 3,7% |
| | MAX POOL | 52 | 52 | 256 | 26 | 26 | 256 | 2 | 2 | 2 | 0 | | - | - | 0,00% | 173.056,00 | 0,660 | 0,9% |
| | CONV9 | 26 | 26 | 256 | 26 | 26 | 512 | 3 | 3 | 1 | 1 | 512 | 1.180.160,00 | 4,502 | 1,76% | 346.112,00 | 1,320 | 1,8% |
| | CONV10 | 26 | 26 | 512 | 26 | 26 | 256 | 1 | 1 | 1 | 0 | 256 | 131.328,00 | 0,501 | 0,20% | 173.056,00 | 0,660 | 0,9% |
| | CONV11 | 26 | 26 | 256 | 26 | 26 | 512 | 3 | 3 | 1 | 1 | 512 | 1.180.160,00 | 4,502 | 1,76% | 346.112,00 | 1,320 | 1,8% |
| | CONV12 | 26 | 26 | 512 | 26 | 26 | 256 | 1 | 1 | 1 | 0 | 256 | 131.328,00 | 0,501 | 0,20% | 173.056,00 | 0,660 | 0,9% |
| | CONV13 | 26 | 26 | 256 | 26 | 26 | 512 | 3 | 3 | 1 | 1 | 512 | 1.180.160,00 | 4,502 | 1,76% | 346.112,00 | 1,320 | 1,8% |
| | MAX POOL | 26 | 26 | 512 | 13 | 13 | 512 | 2 | 2 | 2 | 0 | | - | - | 0,00% | 86.528,00 | 0,330 | 0,5% |
| | CONV14 | 13 | 13 | 512 | 13 | 13 | 1024 | 3 | 3 | 1 | 1 | 1024 | 4.719.616,00 | 18,004 | 7,03% | 173.056,00 | 0,660 | 0,9% |
| | CONV15 | 13 | 13 | 1024 | 13 | 13 | 512 | 1 | 1 | 1 | 0 | 512 | 524.800,00 | 2,002 | 0,78% | 86.528,00 | 0,330 | 0,5% |
| | CONV16 | 13 | 13 | 512 | 13 | 13 | 1024 | 3 | 3 | 1 | 1 | 1024 | 4.719.616,00 | 18,004 | 7,03% | 173.056,00 | 0,660 | 0,9% |
| | CONV17 | 13 | 13 | 1024 | 13 | 13 | 512 | 1 | 1 | 1 | 0 | 512 | 524.800,00 | 2,002 | 0,78% | 86.528,00 | 0,330 | 0,5% |
| | CONV18 | 13 | 13 | 512 | 13 | 13 | 1024 | 3 | 3 | 1 | 1 | 1024 | 4.719.616,00 | 18,004 | 7,03% | 173.056,00 | 0,660 | 0,9% |
| DETETION HEAD | CONV19 | 13 | 13 | 1024 | 13 | 13 | 1024 | 3 | 3 | 1 | 1 | 1024 | 9.438.208,00 | 36,004 | 14,06% | 173.056,00 | 0,660 | 0,9% |
| | CONV20 | 13 | 13 | 1024 | 13 | 13 | 1024 | 3 | 3 | 1 | 1 | 1024 | 9.438.208,00 | 36,004 | 14,06% | 173.056,00 | 0,660 | 0,9% |
| | PASSTHROUGH CONV13 | 26 | 26 | 512 | 13 | 13 | 2048 | | | | | | | | 0,00% | - | - | 0,0% |
| | CONCAT (CONV20, PASSTH..) | 13<br>13 | 13<br>13 | 1024<br>2048 | 13 | 13 | 3072 | | | | | | - | - | 0,00% | - | - | 0,0% |
| | CONV21 | 13 | 13 | 3072 | 13 | 13 | 1024 | 3 | 3 | 1 | 1 | 1024 | 28.312.576,00 | 108,004 | 42,18% | 173.056,00 | 0,660 | 0,9% |
| | CONV22 | 13 | 13 | 1024 | 13 | 13 | 125 | 1 | 1 | 1 | 0 | 125 | 128.125,00 | 0,489 | 0,19% | 21.125,00 | 0,081 | 0,1% |
| | | | | | | | | | | | | | **67.127.293,00** | **256,07** | **100%** | **18.797.701,00** | **71,71** | **100%** |

Tal y como se hace en YOLO original, primero de realiza un preentrenamiento del *backbone* de clasificación, añadiendo a la Darknet 19 una capa convolucional, un *average pool* y una función *softmax*.

Se usa el dataset ImageNet 1000. Se entrenan 160 *epochs* a la resolución de 224×224 y finalmente otras 10 *epochs* a la resolución 448×448.

Para el entrenamiento de detección de objetos se usan 3 capas convolucionales de dimensiones 3×3 de 1024 filtros y una última de 1×1 con el número de filtros coincidente con el número de salidas que queremos para la detección, en este caso 5 cajas delimitadoras con 5 coordenadas cada una y 20 detecciones. Esto es diferente que en YOLO V1, donde la predicción de clase era la misma para todas las cajas dentro de la misma celda. En este caso cada caja delimitadora (5 por celda) tiene su predicción de coordenadas (4), probabilidad de tener un objeto (1) y la predicción de clase (20), en total salida de cada ceda es de 5×25 = 125.

### 2.3.6 YOLO V3 [27]:

En esta versión se reemplazan las capas *max-pooling* con capas convolucionales con *stride* 2, además se introducen conexiones residuales (como en RESNET). En la Tabla 7 se muestra el modelo *backbone* Darknet-53 utilizado para la extracción de características de la imagen, las ultimas capas de *Avgpool*, *Fully Connected* y *Softmax*, se utilizan solo durante el proceso de entrenamiento de características con un *dataset* de clasificación, según la descripción de la arquitectura extraída de la referencia.[27]

*Tabla 7: Arquitectura Darknet-53 para la extracción de características de la imagen en YOLOV3*

| | Tipo | nº de filtros | Kernel/stride | Salida |
|---|---|---|---|---|
| | Convolucional | 32 | 3x3 | 256x256 |
| | Convolucional | 64 | 3x3/2 | 128x128 |
| | Convolucional | 32 | 1x1 | |
| 1x | Convolucional | 64 | 3x3 | |
| | Residual | | | 128x128 |
| | Convolucional | 128 | 3x3/2 | 64x64 |
| | Convolucional | 64 | 1x1 | |
| 2x | Convolucional | 128 | 3x3 | |
| | Residual | | | 64x64 |
| | Convolucional | 256 | 3x3/2 | 32x32 |
| | Convolucional | 128 | 1x1 | |
| | Convolucional | 256 | 3x3 | |
| | Residual | | | 32x32 |
| | Convolucional | 512 | 3x3/2 | 16x16 |
| | Convolucional | 256 | 1x1 | |
| 8x | Convolucional | 512 | 3x3 | |
| | Residual | | | 16x16 |
| | Convolucional | 1024 | 3x3/2 | 8x8 |
| | Convolucional | 512 | 1x1 | |
| 8x | Convolucional | 1024 | 3x3 | |
| | Residual | | | 8x8 |
| | Avgpool | | Global | |
| | T. Conectada | | 1000 | |
| | Softmax | | | |

Se introduce un elemento nuevo a la salida del *backbone* llamado *Spatial Pyramid Pooling* (SPP), que concatena varios *maxpooling* sin hacer submuestreo (*stride* = 1), cada uno con tamaño de *kernel* distinto, permitiendo un mayor campo de recepción. Según el artículo original mejora el $AP_{50}$ en un 2.7%.

Se realiza una predicción multiescala que permite tener diferentes tamaños de celda, basado en Feature Pyramid Network [29]. El objetivo es mejorar la predicción de objetos pequeños. La primera salida es un tensor 13x13x[3x(4+1+80)], la segunda salida es resultado de la concatenación de la primera tras un sobremuestreo con la salida Res8 obteniéndose un tensor 26x26x[3x(4+1+80)], la tercera salida seguirá el mismo procedimiento y el tensor será 52x52x[3x(4+1+80)]. Donde el 4 son las coordenadas de cada caja delimitadora, 1 la predicción de que haya un objeto en la caja y 80 el número de categorías al usar el *dataset* COCO.

La Figura 23 muestra la arquitectura YOLO V3. A destacar la introducción del *Spatial Pyramid Polling* a la salida del *backbone*, así como la predicción multiescala con celdas de 13×13, 26×26 y 52×52.

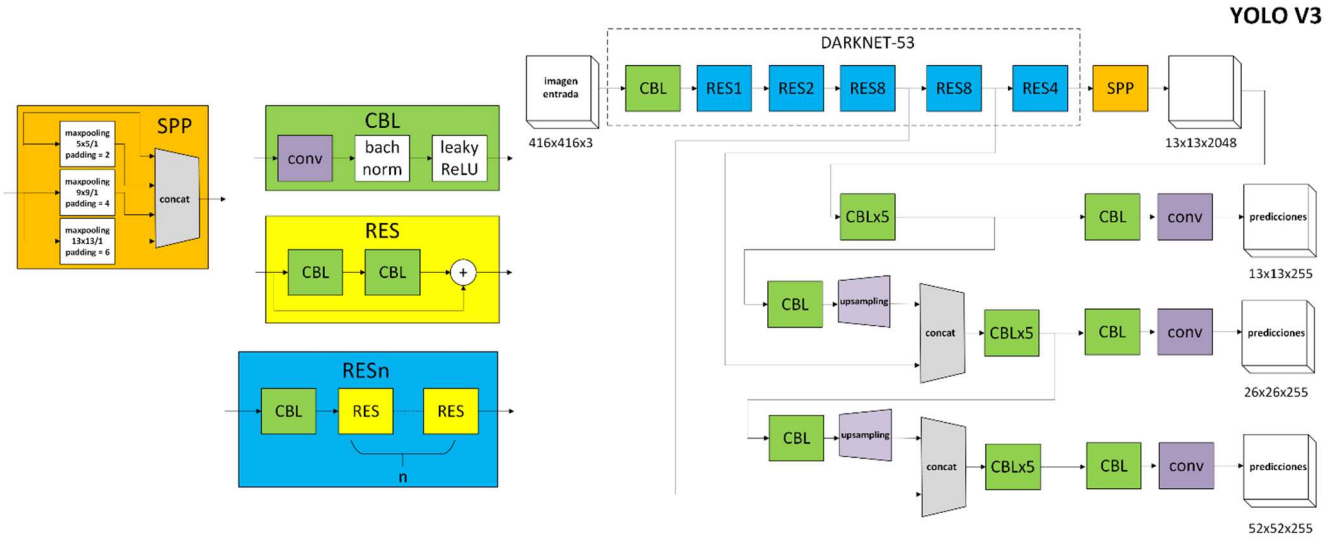



*Figura 23: Arquitectura YOLO V3. Diagrama basado en [16]*

Para la predicción de clase no utilizará *softmax*, en su lugar utilizará un clasificador logístico independiente para cada clase, que entrenará con la función de pérdida *Binary Cross Entropy*, de modo que puede tener múltiples etiquetas en la misma caja delimitadora. Es decir, hace una predicción binaria independiente de cada clase dentro del *Bounding Box*.

### 2.3.7 YOLO V4 [28]

Esta nueva versión es propuesta por Alexey Bochkovskiy, Chien-Yao Wang y Hong-Yuan Mark Liao en [28], diferentes autores a las versiones anteriores, pero aun así es presentada como YOLO dado que mantenía la misma filosofía.

No aparecen cambios significativos en la arquitectura respecto a la versión anterior, esta versión está enfocada a optimizar experimentando con muchos cambios. Se distingue en el trabajo entre *'bag-of-freebies'* y *'bag-of-specials'*.

*Bag-of-freebies:* Son métodos que solo cambian la estrategia de entrenamiento y aumentan el costo de entrenamiento, pero sin aumentar el tiempo de inferencia; el más común es utilizar *data augmentation*. Como novedad se introducen imágenes en mosaico que combinan cuatro imágenes en una sola, lo que permite la detección de objetos fuera de su contexto habitual y también mejora la normalización con lotes pequeños de imágenes de entrenamiento. Se utilizará *DropBlock* [30] en lugar de *Dropout*, más eficaz para redes neuronales convolucionales ya que bloquea regiones contiguas de los mapas de características durante el aprendizaje. También se realizará un suavizado de las etiquetas de clase, entre otras mejoras.

*Bag-of-specials:* Son métodos que centran en el modelo de inferencia. Básicamente se centran en la selección del *backbone*, que terminaría siendo un Darknet53 con conexiones parciales entre etapas (CSPNet)[31], esto reduce el coste computacional mantenido la precisión. Se utiliza la función de activación Mish [32] (como alternativa a ReLU) diferenciable en todo su dominio y no monotónica.

### 2.3.8 YOLO V5 [33]

Desarrollado pocos meses después de YOLO V4, usa las mejoras introducidas en la versión anterior. Como *backbone* utiliza *Cross Stage Partial Darknet* y se implementará usando la librería Pytorch[34]. YOLO V5 no tiene un artículo científico revisado por pares, existe sin embargo un repositorio oficial de GitHub [41]

Respecto a su arquitectura la primera capa (*stem*) tiene un tamaño de *kernel* bastante grande (6×6) y un *stride* de 2, su misión es reducir rápidamente el uso de memoria y el coste computacional. Su arquitectura sigue la filosofía introducida en la versión 3. Se sustituye la SPP (Spatial Pyramid Pooling) por SPPF ('F' de Fast), versión optimizada, que utiliza varias capas secuenciales de *maxpooling* 5×5 en lugar de capas paralelas de diferente tamaño, lo que reduce el coste computacional.

Para el *data augmentation* usara las imágenes en mosaico, transformaciones afines aleatorias, variaciones aleatorias en el espacio de color HSV, etc. como en YOLOV4. Como novedad introduce el *Copy-Paste Augmentation* que consiste en tomar una imagen de objeto y copiarla en una posición aleatoria del fondo de otra imagen. Esto ayuda al modelo a aprender a detectar objetos en contextos y posiciones que no están en el *dataset*. También se introduce el *MixUp* que consiste en combinar dos imágenes y sus etiquetas correspondientes de forma lineal. Realiza una mezcla ponderada de dos imágenes y sus respectivas etiquetas.

La Figura 24 muestra el aumento de datos por copia y pegado. Se trata del método utilizado en YOLOV5 que copia parches aleatorios de una imagen y los pega en otra imagen elegida al azar, generando así una nueva muestra de entrenamiento.


*Figura 24: Aumento da datos por copia y pegado. Imagen obtenida de [42]*

La Figura 25 muestra el aumento de datos *MixUp*: Método utilizado en YOLO v5 para crea imágenes compuestas tomando una combinación lineal de dos imágenes y sus etiquetas asociadas. Además, estas imágenes tienen aplicado aumento en mosaico previamente.

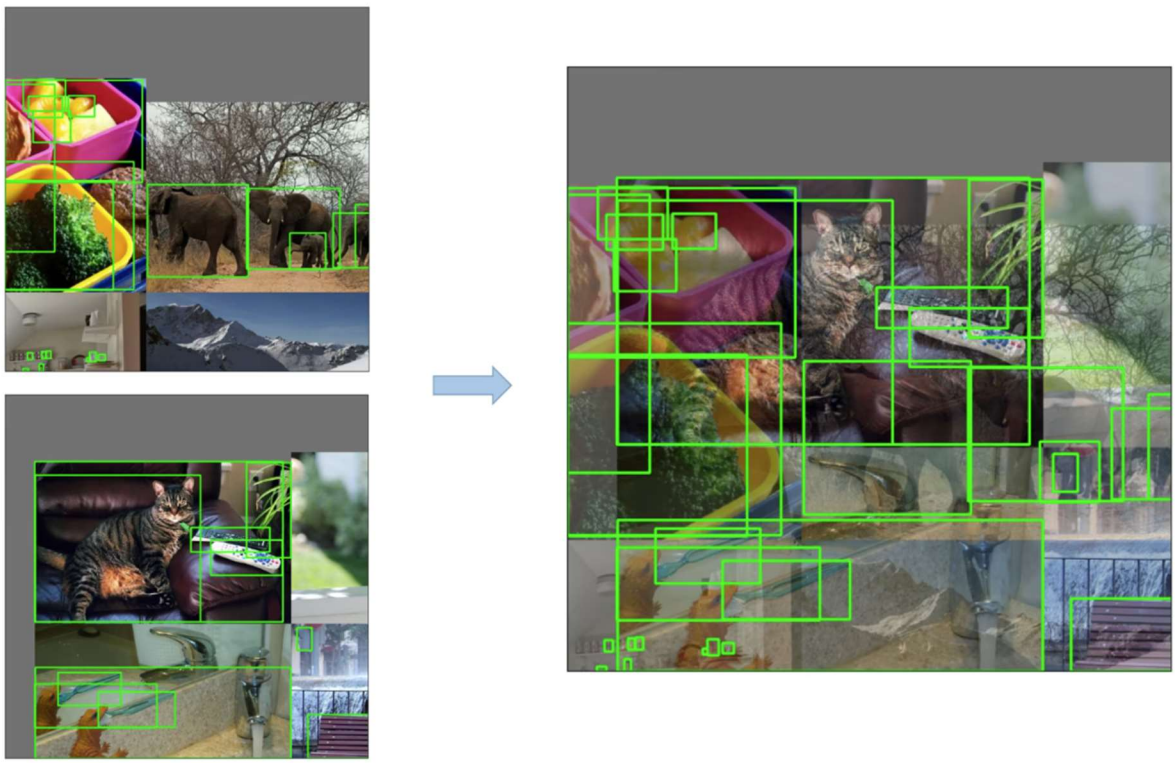


*Figura 25: Aumento de datos por MixUp. Imagen obtenida de [42]*

La Figura 26 muestra la arquitectura de YOLO V5. En ella se incluye un *stem* de tamaño de *kernel* grande (6×6) y un *stride* de 2 para reducir rápidamente la memoria, a continuación utiliza la estructura CSPLayer (*Cross Stage Partial Networks*) representados como C3 en el gráfico, además utiliza PANet (*Path Aggregation Network*) que permite que las características de las capas superiores e inferiores fluyan en ambos sentidos. Usa SPPF (*Spatial Pyramid Pooling Fast*) que utiliza varias capas secuenciales de maxpooling 5×5. Esta versión introduce 5 versiones escaladas: *nano, small, medium, large y extra large*.

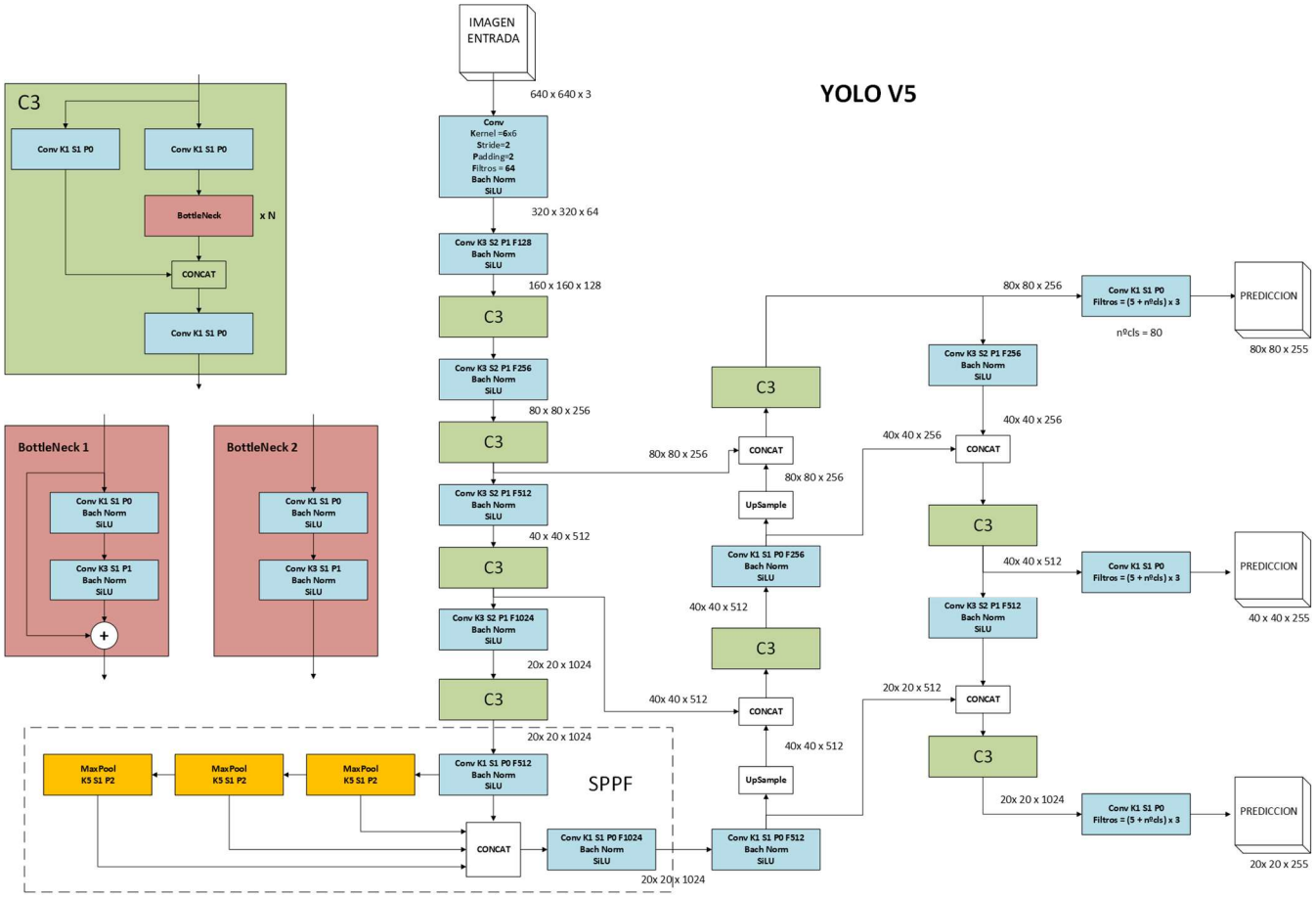



*Figura 26: Arquitectura de YOLO V5. Diagrama basado en [43]*

### 2.3.9 YOLO V6 [35]

Para el *backbone* se utilizará la arquitectura de red neuronal llamada RepVGG, cuya característica principal es la reparametrización estructural, esto le permite transformar estructuras de entrenamiento complejas con conexiones residuales en modelos más sencillos para la inferencia que se convierten básicamente en convoluciones 3x3.

La Figura 27 muestra dos estructuras. La de la izquierda se utilizará durante el entrenamiento, usa una pila de bloques RepVGG con activación ReLU. La de la derecha representa la estructura utilizada para la inferencia: el bloque RepVGG se sustituye por un RepConv.

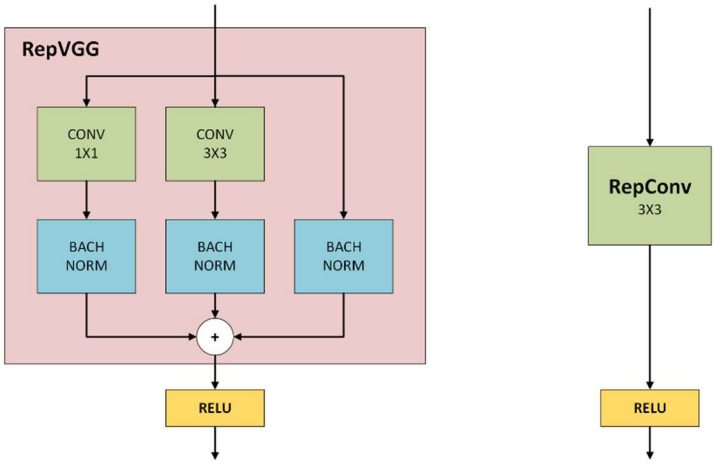



*Figura 27: Bloques RepVGG para entrenamiento y RepConv para inferencia.*

Al igual que YOLO V4 y V5 utiliza en el cuello de la arquitectura con topología PAN '*Path Aggregation Network'* (Red de Agregación de Caminos) que mejora la representación de características en diferentes escalas. Añade un flujo de abajo hacia arriba que reutiliza las características de las capas superiores para fortalecer las inferiores. Esto se consigue creando una estructura mallada donde las características fluyen en ambas direcciones, tal y como puede observarse en la Figura 28.

Respecto a la cabeza de la arquitectura se desacopla la parte de clasificación de la de regresión formando lo que se denomina un cabezal desacoplado

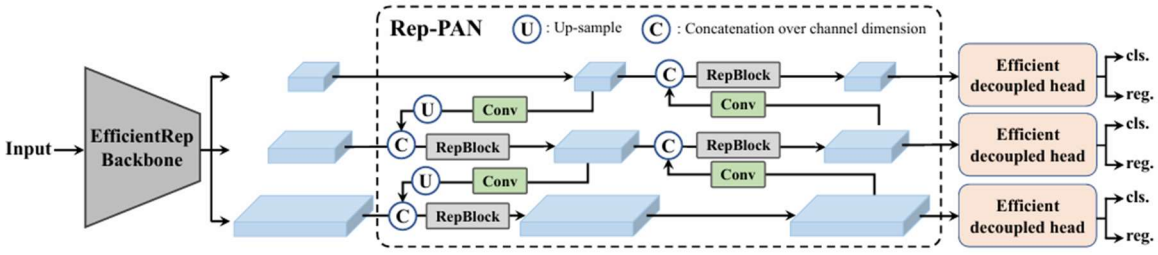



*Figura 28: Arquitectura de YOLO V6. Obtenido de [35]*

### 2.3.10 YOLO V7 [36]

Creada por los mismos autores que YOLO V4 es entrenada usando el MS COCO [25] *dataset* sin entrenamiento previo del *backbone*.

Su arquitectura no supone un cambio significativo respecto de las versiones anteriores, si bien se introduce el módulo E-LAN (*Extended Efficient Layer Aggregation Network*) el cual usa una estrategia de agrupación de características en múltiples escalas.

Se utiliza la re-parametrización como en YOLO V6 pero se utiliza la RepConv sin la conexión identidad lo que fuerza a aprender representaciones más significativas a través de las convoluciones. A la nueva estructura la llaman RepConvN (la N de *no identity*).

### 2.3.11 YOLO V8 [37]

Desarrollada por la compañía que desarrolló YOLO V5, no tiene un artículo científico revisado por pares, existe sin embargo un repositorio oficial de GitHub [44].

Utiliza un backbone muy parecido al de YOLO V5, pero realiza un cambio en las capas CSPLayer, que pasan a llamarse C2f (*cross-stage partial bottleneck with two convolutions*). Esta C2f divide la salida de la primera convolución->bachnormalización->activación SiLU en dos canales, una parte que pasa directa a la concatenación final (*skip conection*) y otra que pasa a uno bloques *bottleneck* (conv 3×3 -> conv 3×3 y cortocircuito), se concatena la mitad que pasa directa con la salida del *bottleneck* para alimentar la convolución final, finalmente habrá una capa de normalización por lotes y activación SiLU. Ver Figura 29.

Es un modelo sin anclajes para cajas delimitadoras y la cabeza del modelo está desacoplada, separando las tareas de detección de las de clasificación.

Utiliza la función de activación sigmoide para la predicción de las coordenadas de la caja delimitadora, para la estimación de la confianza de que haya un objeto en la caja delimitadora y para la clasificación, por lo que el modelo predice la probabilidad de pertenecer a cada una de las clases de manera independiente (esto es común desde la versión 3).

Para la función de pérdida de las cajas delimitadoras se utiliza CIoU [38] (*Complete Interection Over Unión*) que es una mejora de la métrica IoU, ya que además tiene en cuenta la distancia entre los centros de las cajas y la razón de aspecto de las mismas, y DFL[39] (*Distribution Focal Loss*) donde cada coordenada se representa como una distribución sobre múltiples puntos discretos, de modo que se predicen las coordenadas como distribuciones discretas. Para la función de pérdida de clasificación se utiliza BCE (*Binary Cross Entropy*) por lo que calcula la pérdida de manera independiente para cada clase.

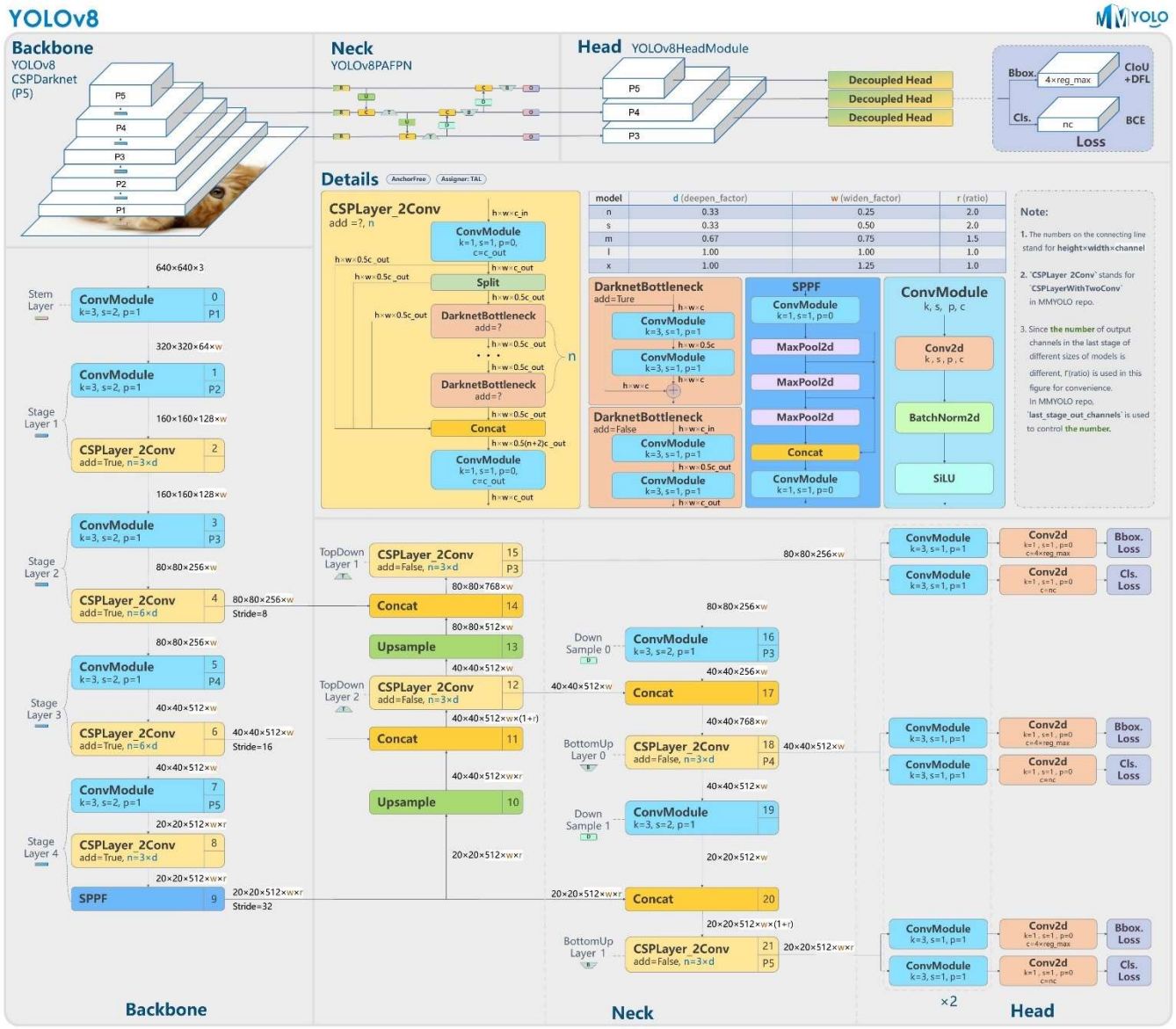



*Figura 29: Arquitectura YOLO V8. Representación gráfica obtenida de [45]*

# Capítulo 3

# Materiales y métodos

Se describen los materiales y métodos utilizadas para la realización de este trabajo, tanto a nivel de hardware como de software.

## 3.1. Materiales

### 3.1.1. Software utilizado

En esta sección se describen los recursos y herramientas utilizadas para la exploración de las capacidades de la librería YOLOv8 en la detección y clasificación de objetos en imágenes.

a) **Roboflow**

Roboflow es una plataforma utilizada para la gestión, procesamiento y aumento de *datasets* de visión por computador. En este trabajo, se ha empleado para la anotación y preprocesamiento de datos, permitiendo la generación de conjuntos de entrenamiento compatibles con YOLOv8. Las funcionalidades utilizadas incluyen la conversión de formatos de anotación, normalización de imágenes. Sin embargo, no se han utilizado las capacidades de roboflow para realizar *data augmentation* mediante transformaciones geométricas, ajustes de color, entre otras técnicas. Esa etapa se abordará utilizando las herramientas de aumento de datos que proporciona YOLO: entre otras modificaciones de los canales HSV, giros, traslaciones, cambios de escala, recortes, mosaico, mezcla de imágenes, etc.

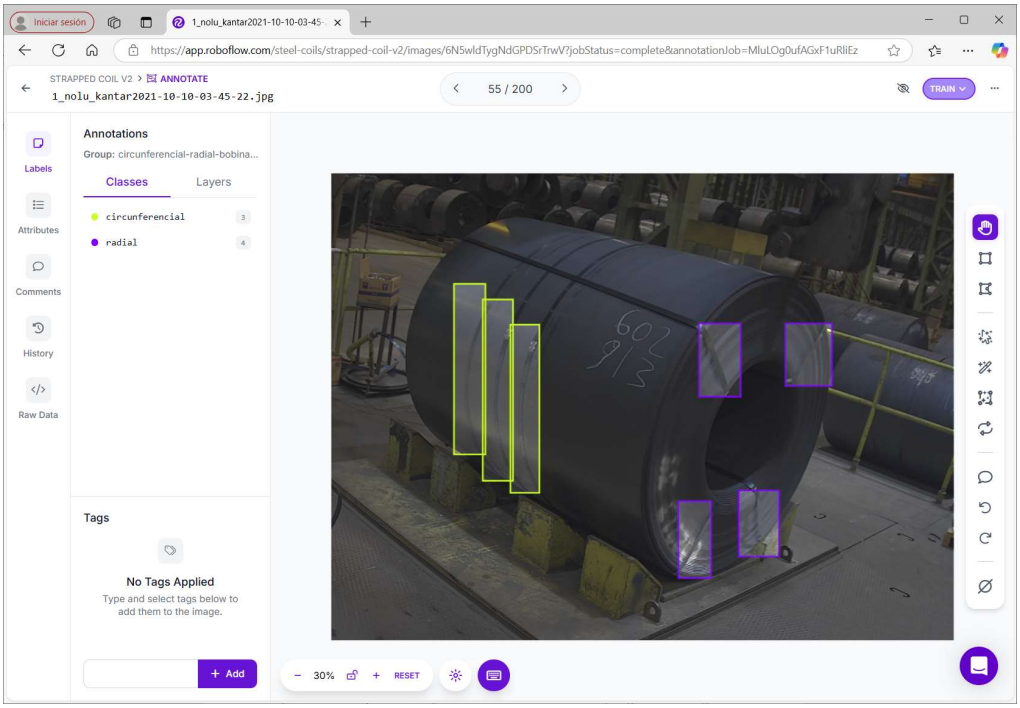



*Figura 30: Ejemplo de etiquetado en Roboflow.*

b) Intérprete **Python 3.9.19**

El entorno de desarrollo se basa en Python 3.9.19, una versión que garantiza compatibilidad con las librerías utilizadas. Se elige Python por su ecosistema maduro en inteligencia artificial y aprendizaje profundo, además de su compatibilidad con bibliotecas optimizadas para cálculo numérico y aceleración por hardware.

c) Gestor de paquetes: **PIP**

Para la gestión e instalación de dependencias, se ha empleado pip, el gestor de paquetes de Python. Se utilizó para instalar y actualizar las librerías necesarias, asegurando la integración de las últimas versiones de YOLOv8 y sus dependencias.

d) Librerías **PyTorch**

PyTorch es una biblioteca de aprendizaje profundo desarrollada por Meta AI. En este trabajo, se ha utilizado como *backend* para la ejecución de YOLOv8, aprovechando su capacidad para computación en GPU y su compatibilidad con CUDA. PyTorch proporciona un entorno flexible para la definición y entrenamiento de modelos neuronales.

e) Librería **Ultralytics**

La librería Ultralytics proporciona una implementación optimizada de YOLOv8. Se ha utilizado para la carga del modelo, la inferencia sobre imágenes y la evaluación del desempeño. Esta librería incluye funciones para el ajuste de hiperparámetros y la visualización de resultados.

f) Librería **OpenCV**

OpenCV es una biblioteca de procesamiento de imágenes utilizada para la manipulación de datos de entrada. En este estudio, se ha empleado para la lectura, preprocesamiento y transformación de imágenes antes de ser ingresadas en el modelo YOLOv8. Se han utilizado funciones para el escalado, conversión de color, aplicación de filtros y transformaciones geométricas.

g) Librería **NumPy**

NumPy es una biblioteca fundamental para la computación científica en Python, proporcionando soporte para arrays multidimensionales y funciones matemáticas optimizadas. En este trabajo, se ha empleado para la manipulación y preprocesamiento de datos numéricos, facilitando operaciones como la normalización de imágenes y el manejo eficiente de datos estructurados antes de su entrada en el modelo YOLOv8.

h) Aceleración por hardware usando **GPU: CUDA**

CUDA es una API desarrollada por NVIDIA que permite la ejecución paralela de algoritmos en GPU. En este trabajo, se ha empleado para mejorar el rendimiento del entrenamiento con YOLOv8, reduciendo significativamente los tiempos de procesamiento en comparación con la ejecución en CPU.

i) Dataset **COCO 2017**

El dataset utilizado para la evaluación del modelo es COCO 2017, un conjunto de datos extenso y estandarizado en visión por computador. Contiene anotaciones de objetos en diversas categorías, permitiendo la evaluación del desempeño de YOLOv8 en diferentes escenarios de detección.

j) Editor de código fuente: **Visual Studio Code (Microsoft)**

Visual Studio Code (VS Code) ha sido utilizado como editor de código fuente en este trabajo. Se trata de un entorno de desarrollo ligero pero potente, que ofrece compatibilidad con múltiples lenguajes y extensiones. Se ha utilizado por su integración con Python y su facilidad para la gestión de entornos virtuales. Además, sus herramientas de depuración han facilitado el desarrollo y prueba del código.

k) Entornos virtuales: **Conda**

Para la gestión de entornos virtuales, se ha utilizado Conda, una herramienta que permite la creación y administración de entornos aislados con versiones específicas de Python y paquetes. Conda ha facilitado la instalación de dependencias y la compatibilidad entre librerías, asegurando la reproducibilidad del entorno de desarrollo y evitando conflictos entre paquetes.

### 3.1.2. Hardware utilizado

Los recursos utilizados son un ordenador personal con 8Gb de memoria RAM, un procesador INTEL i7-7700 y una GPU Nvidia GTX1050Ti 4Gb con soporte CUDA. Para el entrenamiento de los modelos se ha utilizado la GPU y se ha limitado el tamaño de los lotes (*batch*) al máximo que permitía usar solo memoria de la GPU.

### 3.1.3. Imágenes utilizadas

Las imágenes empleadas en la prueba del desempeño de YOLOv8 están bajo la licencia Creative Commons Attribution 4.0 International (CC BY 4.0). Esta licencia permite el uso, distribución, modificación y adaptación de los datos, siempre que se otorgue el crédito adecuado a los autores originales. Desde el punto de vista técnico, esta licencia garantiza la disponibilidad de los datos para investigaciones y aplicaciones sin restricciones en su reutilización, con la única condición de atribución. Esto facilita la reproducibilidad de experimentos y el desarrollo de modelos en entornos académicos e industriales.

CircleDetector Dataset, Open Source Dataset, author Ömer Can, License: CC BY 4.0, https://universe.roboflow.com/omer-can/circledetector, Marzo 2022.

La Figura 31 muestra la generación del *dataset* para YOLOv8 desde la aplicación de Roboflow. Aparte del etiquetado de las imágenes, el único preprocesamiento que se hace de estas es el ajustar las imágenes a una resolución de 640×640. No se aplica ninguna técnica de aumento de datos, dado que esta parte se abordará con YOLOv8.

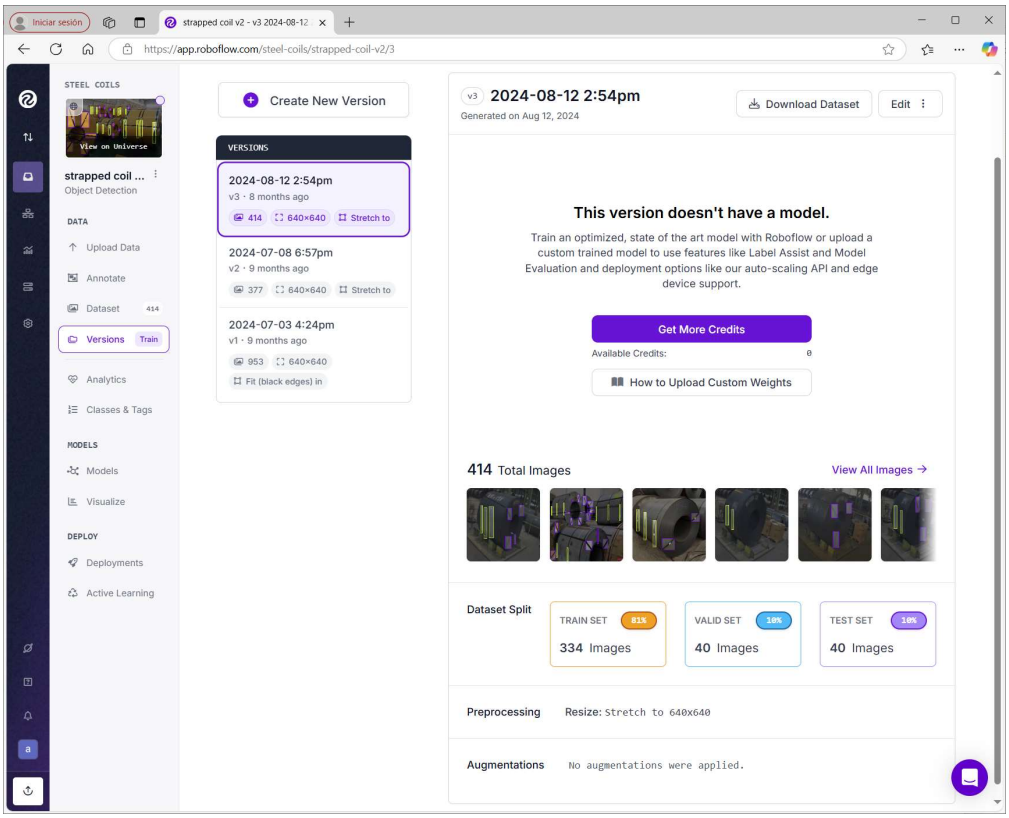



*Figura 31: Generación del data set para YOLOV8 desde la aplicación de Roboflow.*

Tal y como puede observarse en la Figura 32, el *dataset* sólo contiene 414 imágenes y 1984 anotaciones. Se reservan el 10% de las imágenes para validación y otro 10% para test.

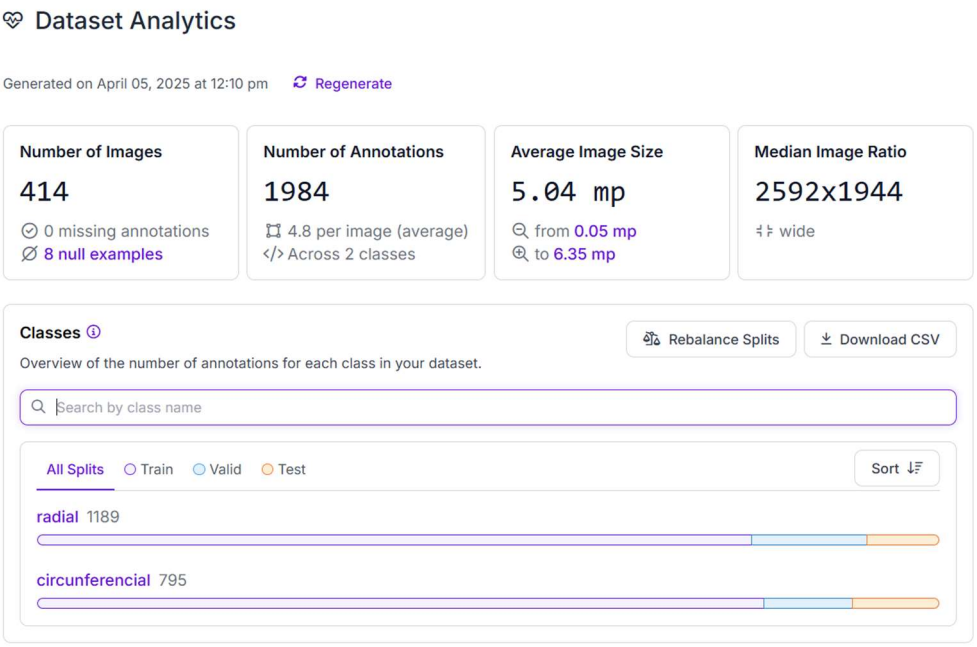



*Figura 32: Análisis del conjunto de datos mostrado por Roboflow.*

## 3.2. Métodos.

Utilizando las imágenes obtenidas de la plataforma Roboflow (2024) [7], en total 314 imágenes captadas por una cámara fija sobre una estación de flejado manual y añadiendo otras 100 imágenes disponibles en internet sin copyright, con distintas perspectivas y con la presencia de varias bobinas vamos a entrenar una red neuronal convolucional de detección y clasificación de objetos.

Primero se anotan todos los elementos presentes en las imágenes, este trabajo supone una cantidad de tiempo importante que no tiene un valor académico ni de aprendizaje para el alumno, pero que es necesaria para entrenar el modelo. En la Figura 33 se muestran algunas de las imágenes anotadas, del total de 414 imágenes.

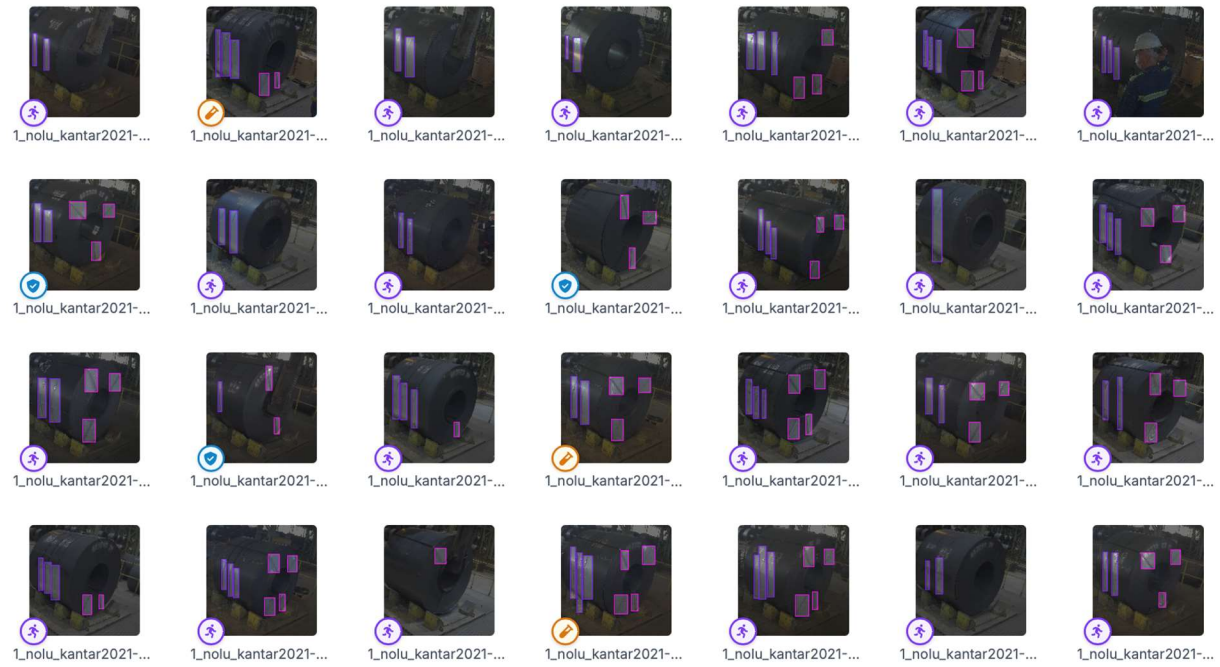



*Figura 33: Anotaciones en Roboflow del dataset de bobinas flejadas [7] con licencia CC BY 4.0.*

El conjunto de imágenes del *dataset* se distribuye en 80% para entrenamiento, 10% para validación del modelo y otro 10% para test.

Aunque el objetivo es detectar los flejes vistos desde una cámara fija, se utilizan también imágenes de bobinas desde diferentes posiciones de captura para detectar el fleje desde otras perspectivas, con la intención de conseguir un modelo de detección más robusto.

Sin embargo, el objetivo es detectar los flejes de una bobina vista siempre con la misma perspectiva, dado que la cámara está fija y las bobinas se sitúan siempre en la misma posición. Esto es habitual en instalaciones industriales, donde existen cámaras para controlar los procesos. Se entrena un modelo de detección de objetos, que permitirá distinguir entre dos clases o tipos de flejes: radiales y circunferenciales. La imagen de la Figura 3, muestra, con carácter ilustrativo, el resultado de la detección sobre una imagen reservada para test.

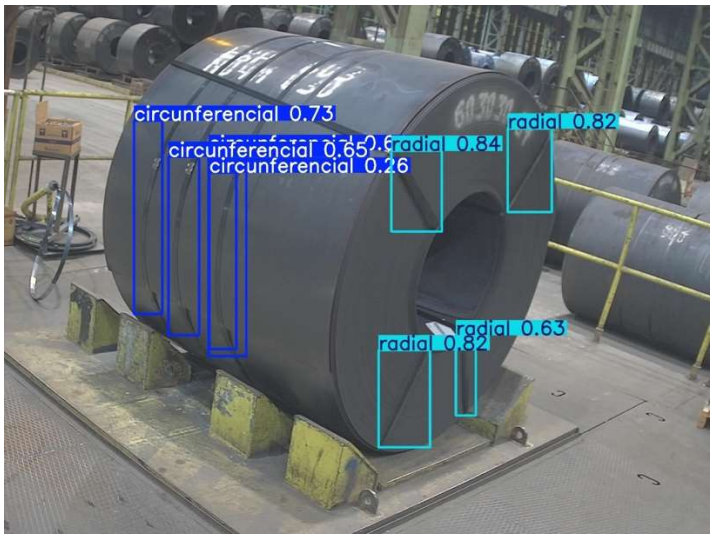



*Figura 34: Detección de objetos y confiablidad en la predicción con modelo yolov8n.*

Interesa también distinguir entre las diferentes partes de la bobina, por lo que se utiliza un modelo de detección de objetos en imágenes, con ello se pretende identificar tres clases de estructuras, a saber, Figura 35: Parte cilíndrica de la bobina, Cara plana y Agujero de la bobina.

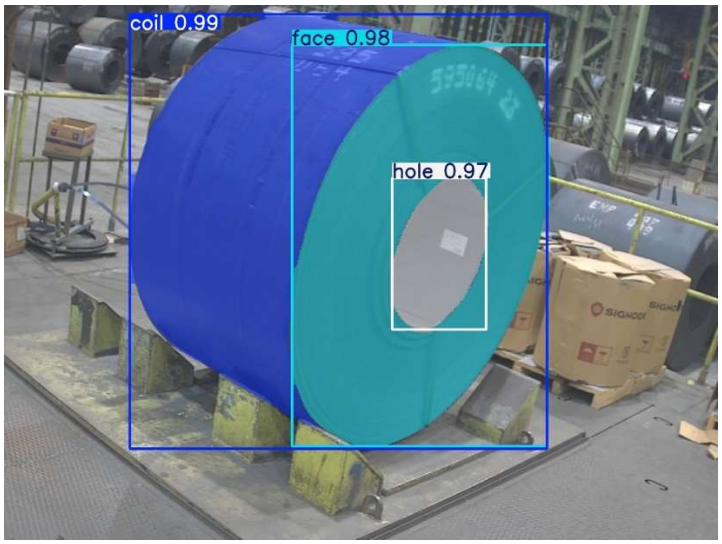



*Figura 35: Detección de objetos y confiablidad en la predicción con modelo yolov8n-seg.*

El siguiente paso consiste en la obtención de una imagen frontal de la bobina utilizando la máscara de la clase ‘face’, que se obtiene del modelo de segmentación de instancias. La intención es que el operador de calidad pueda obtener una imagen frontal de la bobina con la posición de los flejes en disposición radial. Para lo que se han aplicado dos métodos:

1. Encontrar la elipse que mejor se ajusta a la cara de la bobina y después utilizando una transformación afín, ecuación (3.1), convertir esta elipse en un círculo. Esto genera una imagen frontal de la bobina, aunque deformada porque no se ha tenido en cuenta la perspectiva, tal y como se muestra en la Figura 36. La imagen de la izquierda muestra la máscara de la clase ‘face’. La imagen de la derecha corresponde a la transformación afín.

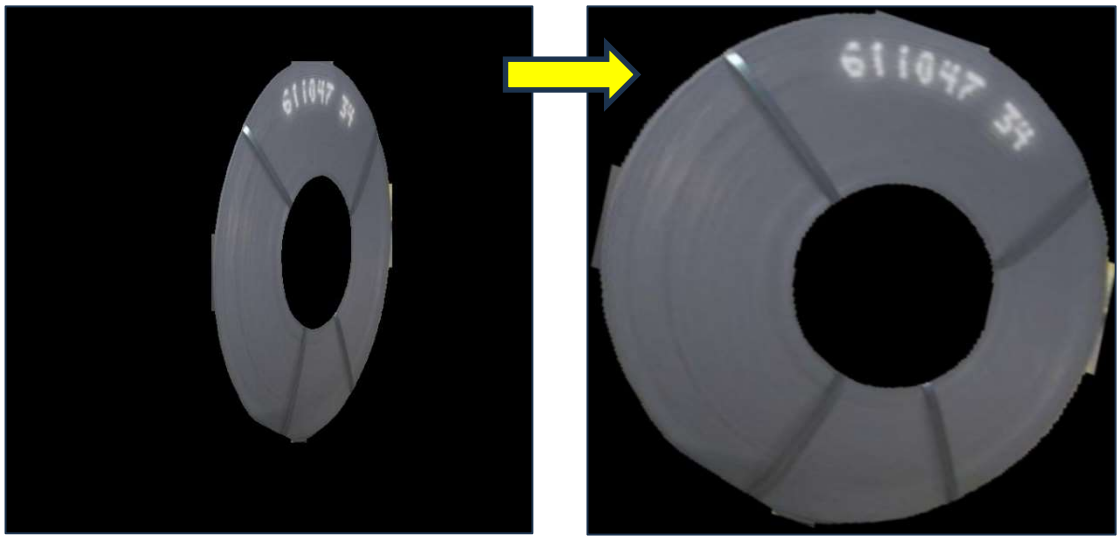


*Figura 36: Ejemplo de transformación afín de la cara de la bobina.*

Para lo cual se genera una función que calcula la matriz afín de 2×3:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & b_{00} \\ a_{10} & a_{11} & b_{10} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad (3.1)$$

2. Calcular la matriz de **transformación en perspectiva** de dimensión 3×3. Para lo cual se buscan los puntos de fuga usando elementos geométricos conocidos en la imagen. En la imagen de la Figura 37, se observa que la cámara no está bien nivelada, dato que la línea de horizonte que une ambos puntos de fuga está inclinada (es fácil apreciarlo mirando los pilares de la nave).

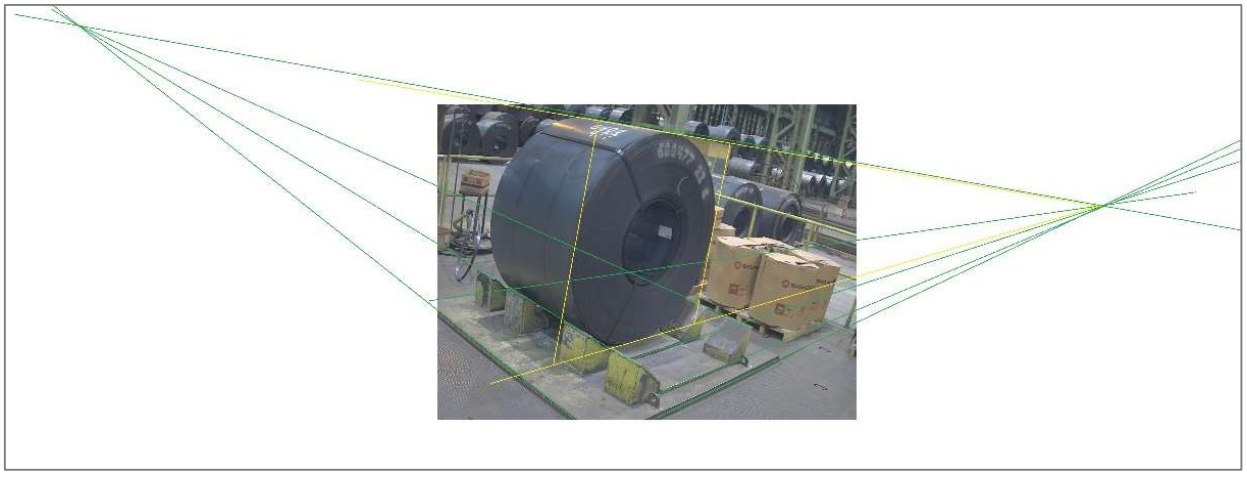


*Figura 37: Identificación de los puntos de fuga.*

Se utilizan cuatro puntos y su posición futura, para calcular la matriz de transformación, para ello se utilizan las funciones que tiene OpenCV2 disponibles. La Figura 38 muestra en la imagen de la izquierda la máscara de la clase 'face', la imagen de la derecha corresponde a la transformación en perspectiva.

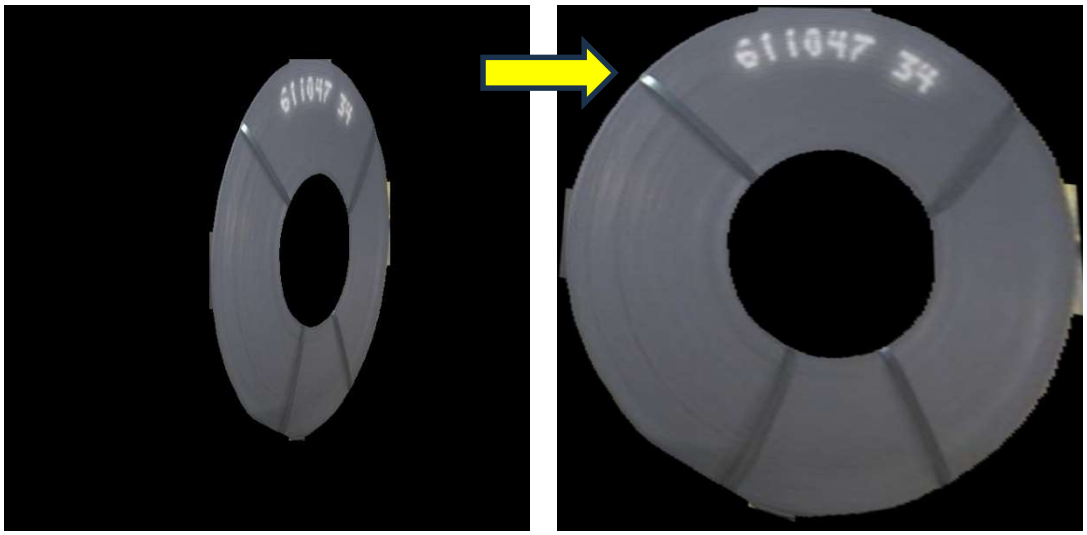


*Figura 38: Ejemplo de transformación en perspectiva de la cara de la bobina.*

El siguiente proceso consiste en identificar la posición angular de los flejes en base a los Boulding Boxes obtenidos con la detección de objetos. La Figura 39 muestra la imagen obtenida de la transformación en perspectiva sobre la máscara de clase 'face'. Se ha dibujado en morado la predicción de la posición del fleje y se inserta un texto identificando el fleje y su posición angular.

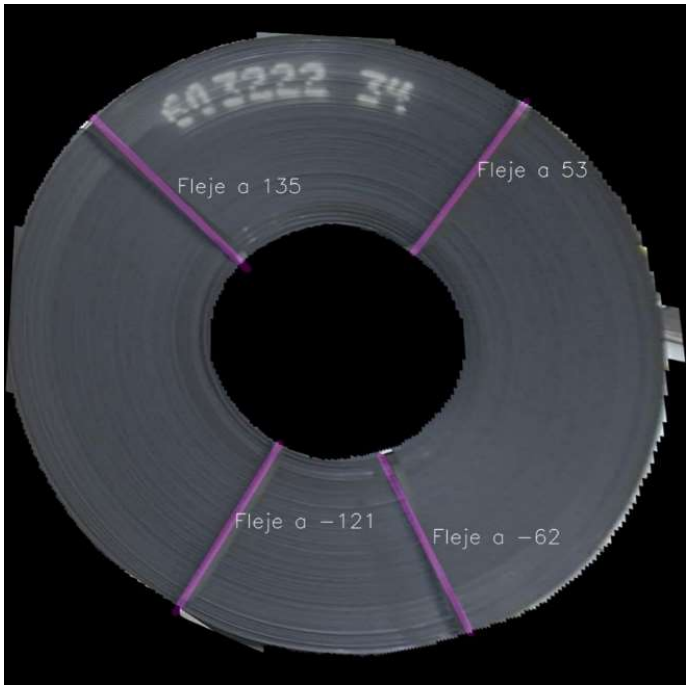



*Figura 39: Predicción de la posición angular del fleje radial.*

# Capítulo 4

# Resultados

Tal y como se describe al principio de esta memoria la idea que motiva este proyecto es la detección de objetos en un entorno industrial utilizando *Deep Learning*. Para este trabajo se utiliza YOLOv8 y se parte siempre de los modelos preentrenados de que dispone la librería mencionada en el capítulo previo.

Dado que no se dispone de una base de datos demasiado extensa se obtienen mejores resultados usando un modelo preentrenado que saca características de las imágenes de entrenamiento de MS COCO [25] que son útiles para la clasificación de nuestros objetos, y que por tanto solo requiere la actualización de las ultimas capas de la red neuronal. Este proceso es lo que se conoce como transferencia de aprendizaje cuyo fundamento estriba en que el modelo tiene aprendidas características generales de los objetos en las primeras capas, tales como bordes, formas, texturas, que se mantienen, de forma que, al cambiar sólo las últimas capas, estas aprenden las novedades específicas del nuevo conjunto de datos, de ahí que con un número reducido sea suficiente.

En la sección 4.1 se compara el rendimiento de los diferentes modelos con la base de imágenes del proyecto. Posteriormente en la sección 4.2 se elige un modelo y sobre este se analiza la influencia de varios tamaños de lote *(batch)*. A continuación, en la sección 4.3 se comparará la convergencia de la función de pérdida a lo largo de las distintas épocas, según la tasa de aprendizaje utilizada. Finalmente, en la sección 4.4 se realiza una transformación en perspectiva de la cara de la bobina para mostrar la posición de los flejes radiales, con el único interés de mostrar unas imágenes visualmente atractivas para un usuario final de control de calidad en un proceso industrial.

La Figura 40 muestra el rendimiento de los modelos de YOLO v8 para detección de objetos, según los resultados proporcionados en [46].

Performance

Detection (COCO) | Detection (Open Images V7) | Segmentation (COCO) | Classification (ImageNet) | Pose (CO >

See Detection Docs for usage examples with these models trained on COCO, which include 80 pre-trained classes.

| Model | size (pixels) | mAP$^{val}$ 50-95 | Speed CPU ONNX (ms) | Speed A100 TensorRT (ms) | params (M) | FLOPs (B) |
|---|---|---|---|---|---|---|
| YOLOv8n | 640 | 37.3 | 80.4 | 0.99 | 3.2 | 8.7 |
| YOLOv8s | 640 | 44.9 | 128.4 | 1.20 | 11.2 | 28.6 |
| YOLOv8m | 640 | 50.2 | 234.7 | 1.83 | 25.9 | 78.9 |
| YOLOv8l | 640 | 52.9 | 375.2 | 2.39 | 43.7 | 165.2 |
| YOLOv8x | 640 | 53.9 | 479.1 | 3.53 | 68.2 | 257.8 |

Figura 40: Rendimiento modelos YOLOV8. Obtenida de [46]

## 4.1. Comparación entre modelos

Se entrenan los distintos modelos que tiene YOLOv8, para ver el desempeño que obtienen con el *dataset* que se ha generado de flejes radiales y circunferenciales. Se mantendrá constante el parámetro de ***batch* = 4** de tal modo que incluso para los modelos más grandes puedan almacenarse el valor de los parámetros, valor de activaciones y gradientes de parámetros en RAM de la GPU para cada iteración (esto hará que el entrenamiento sea más rápido). Se fija un número de épocas ***epochs* = 150** para ver el sobreajuste en los modelos.

El resto de hiperparámetros se mantienen por defecto, a destacar:

**imgsz = 640:** Tamaño de entrada de las imágenes para el modelo. Todas las imágenes se redimensionan a 640×640 píxeles antes de ser procesadas.

**dropout = 0.0:** Proporción de "desactivación" de neuronas durante el entrenamiento para prevenir el sobreajuste. 0.0 significa que no se aplica dropout.

**lr0 = 0.01:** Tasa de aprendizaje inicial. Define los pasos que da el modelo al ajustar sus pesos al comienzo del entrenamiento.

**lrf = 0.01:** Factor de reducción de la tasa de aprendizaje al final del entrenamiento. Controla cómo disminuye el lr0 durante el entrenamiento.

**momentum = 0.937:** Valor que ayuda a acelerar el optimizador en la dirección correcta, suavizando las actualizaciones del gradiente, pues acumula gradientes anteriores.

**warmup_epoch = 3.0:** Número de épocas durante las cuales la tasa de aprendizaje sube progresivamente hasta su valor objetivo, esto ayuda a que el entrenamiento sea más estable al inicio.

**warmup_momentum = 0.8:** Valor inicial del momentum durante la fase de warmup, antes de llegar al valor objetivo.

**warmup_bias_lr =0.1:** Tasa de aprendizaje inicial específica para los biases de las capas durante la fase de warmup.

**optimizer = auto:** El optimizador que se usa para entrenar el modelo. YOLOv8 elige automáticamente el más adecuado generalmente SGD o Adam dependiendo del contexto.

Se observan los resultados en la Figura 41. El gráfico de la izquierda muestra que el modelo 'nano' obtiene mejores métricas tanto en mAP50% como en mAP50%-95% sobre datos de test. Esto es debido a que el conjunto de datos es muy sencillo, con solo 2 clases e instancias distinguibles incluso con el modelo más simple. En el gráfico de la derecha vemos la complejidad de los modelos. En la columna naranja el número de parámetros de cada modelo. Se muestra en azul el tiempo (en horas) empleado por el hardware descrito previamente para realizar el entrenamiento de los distintos modelos.

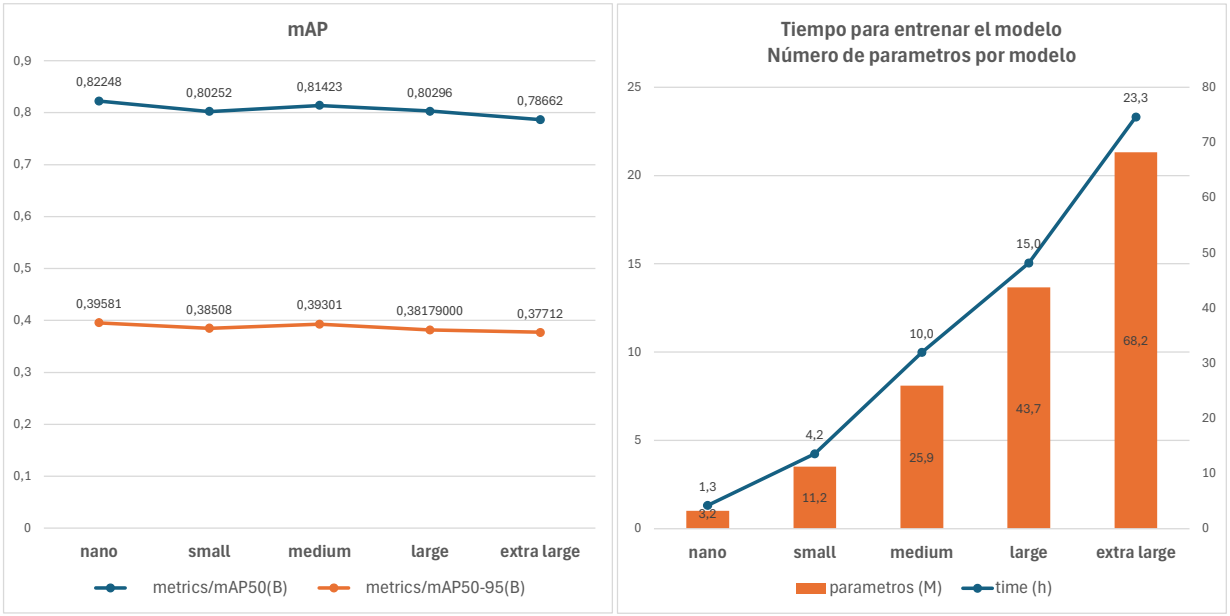



*Figura 41: Rendimiento de los distintos modelos de YOLOv8 en el conjunto de datos del proyecto.*

Se intuye que el mejor desempeño del modelo nano es porque el *conjunto de datos* es pequeño y sencillo, los modelos más complejos deben sufrir mayor sobreajuste, lo comprobamos observando la evolución de la función de pérdida en los datos de validación.

La función de pérdida de YOLOv8 se compone de tres términos:

1. ***Box Loss:*** Que representa la diferencia entre las cajas delimitadoras predichas y las reales, se basa en CIoU. Por tanto, optimiza la precisión de las cajas delimitadoras.

2. ***Cls Loss:*** Evalúa la predicción de clase de cada objeto usando entropía cruzada binaria. Por tanto, mejora la clasificación de los objetos detectados.

3. ***DFL Loss (Distillation Feature Learning):*** Se basa en una regresión discreta de los bordes, para mejorar la ubicación de los bordes. Por tanto, convierte la predicción de bordes en un problema de clasificación discreta sobre bins en lugar de una regresión directa. El objetivo es tener una mayor precisión de los bordes.

Vemos a continuación la evolución de las diferentes componentes de la función de pérdida de los distintos modelos, del *nano* al *extra-large*. Analizando componente a componente de la función de pérdida sobre los datos de validación:

***Box Loss:*** Según la Figura 42, todos los modelos convergen rápido en las primeras 20 épocas. No existe una gran diferencia entre los modelos (todos se estabilizan cerca del valor 1.75), por lo que la complejidad de los modelos no afecta a la calidad de las predicciones. Se aprecia un ligero sobre ajuste, pero no es muy evidente.

El modelo *nano* presenta un desempeño igual que los modelos más complejos. Se observa incluso que en las últimas épocas el modelo nano presenta menor pérdida en *box_loss*.

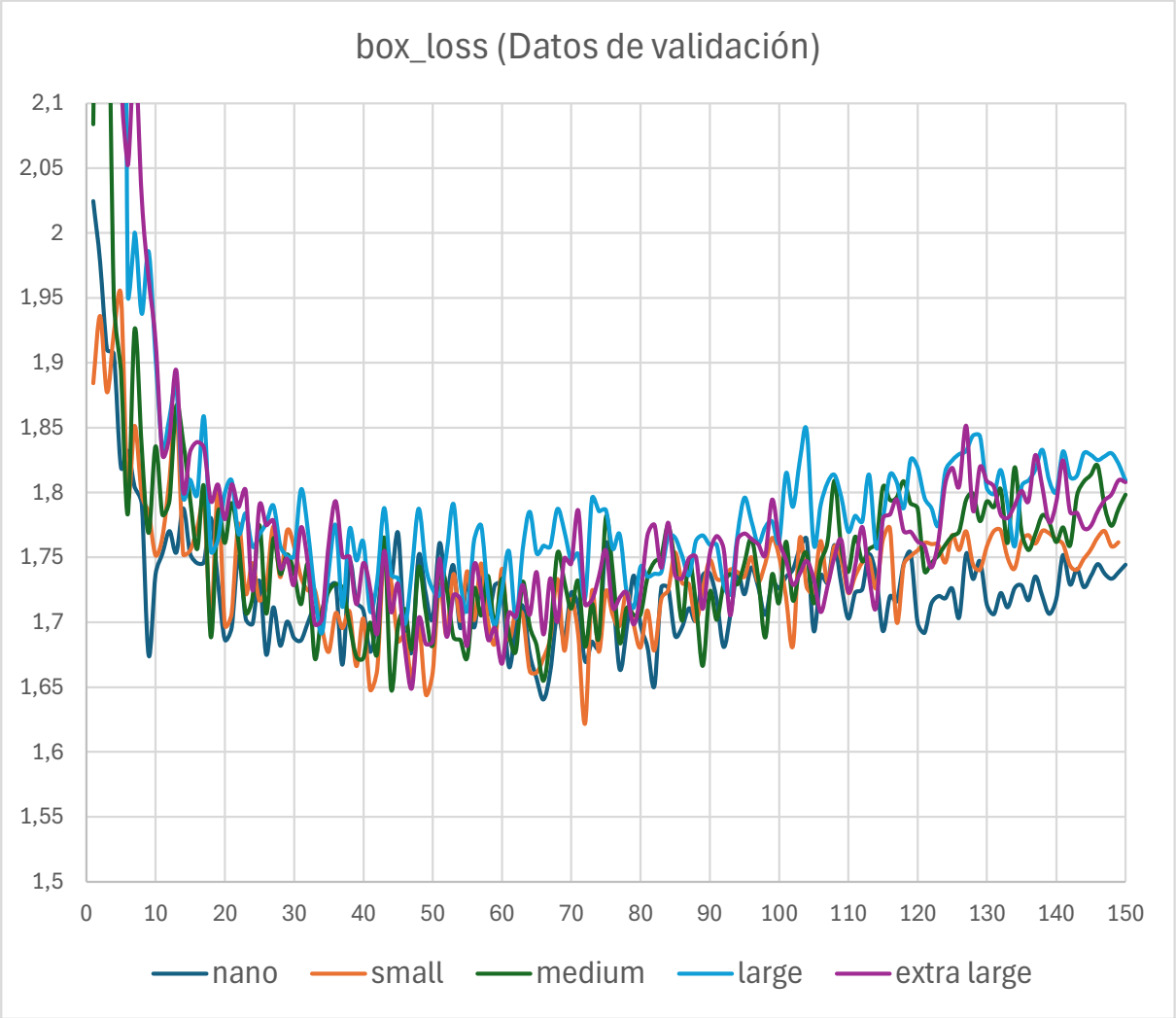



*Figura 42: Evolución de la componente de pérdida box_loss para los diferentes modelos.*

***Cls_Loss:*** En relación a la Figura 43, aparece una disminución clara en las primeras 30 épocas y después todos los modelos se estabilizan entre 0.8 y 1. No se aprecia diferencia entre los modelos y todos aprenden a clasificar.

Ninguno de los modelos presenta sobreajuste y la generalización en la predicción de clase es buena independientemente del modelo utilizado.

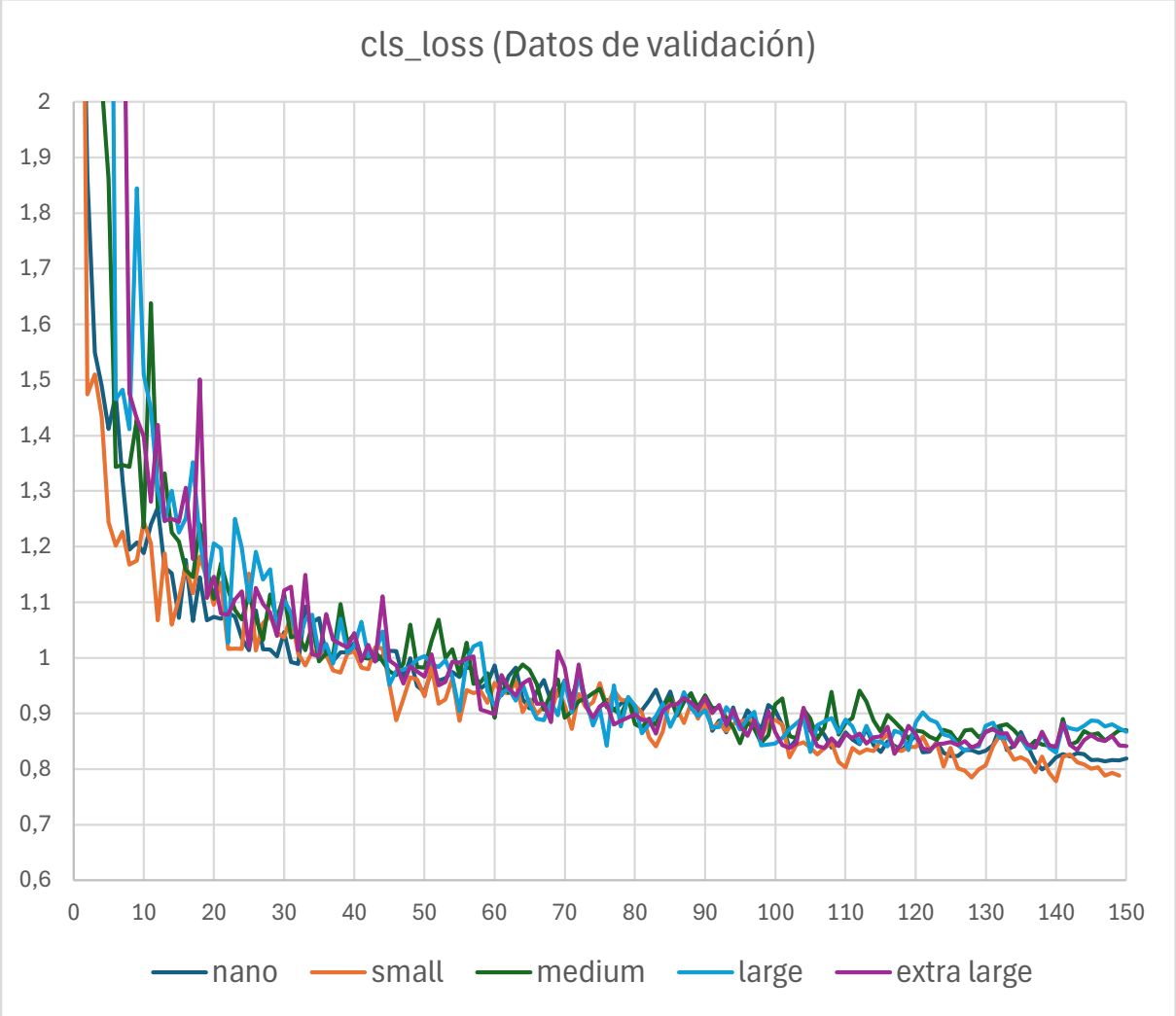



*Figura 43: Evolución de la componente de pérdida cls_loss para los diferentes modelos*

***DFL_Loss:*** En la Figura 44 se observa claramente cómo el modelo *nano* y *small* tienen un valor de pérdida más bajo, mientras que el modelo *large* y *extra large* parten de un valor de pérdida más alto que tiende a crecer sobre la época 70, indicando un claro fenómeno de sobreajuste.

Se observa un claro sobreajuste sobre todo de los modelos más complejos. Esto tiene mucho sentido porque estos modelos realizan una predicción con distribuciones más estrechas y confiadas sobre los *bins*. Como los datos de entrenamiento no son muy confiables, dado que la delimitación de parte del fleje es bastante imprecisa, los modelos más complejos intentan ajustar sobre un *Ground Truth* inexacto y difuso. Esto se debe a que a la hora de etiquetar los flejes circunferenciales no está claro cómo definir sus límites, se describe con más detalle esta problemática en la sección *5.1 de Conclusiones*, concretamente en la Figura 54.

Dado que el etiquetado de los flejes circunferenciales en el conjunto de entrenamiento no es del todo preciso, los modelos más complejos tienden a ajustarse a errores y variabilidad del *Ground Truth*. Por el contrario, el modelo nano realiza predicciones más difusas, generando distribuciones más amplias y menos confiadas, lo que le permite ser más robusto frente a un etiquetado impreciso. Como resultado, el modelo nano mantiene una pérdida más baja y estable a lo largo del entrenamiento, evitando el sobreajuste observado en modelos de mayor tamaño.

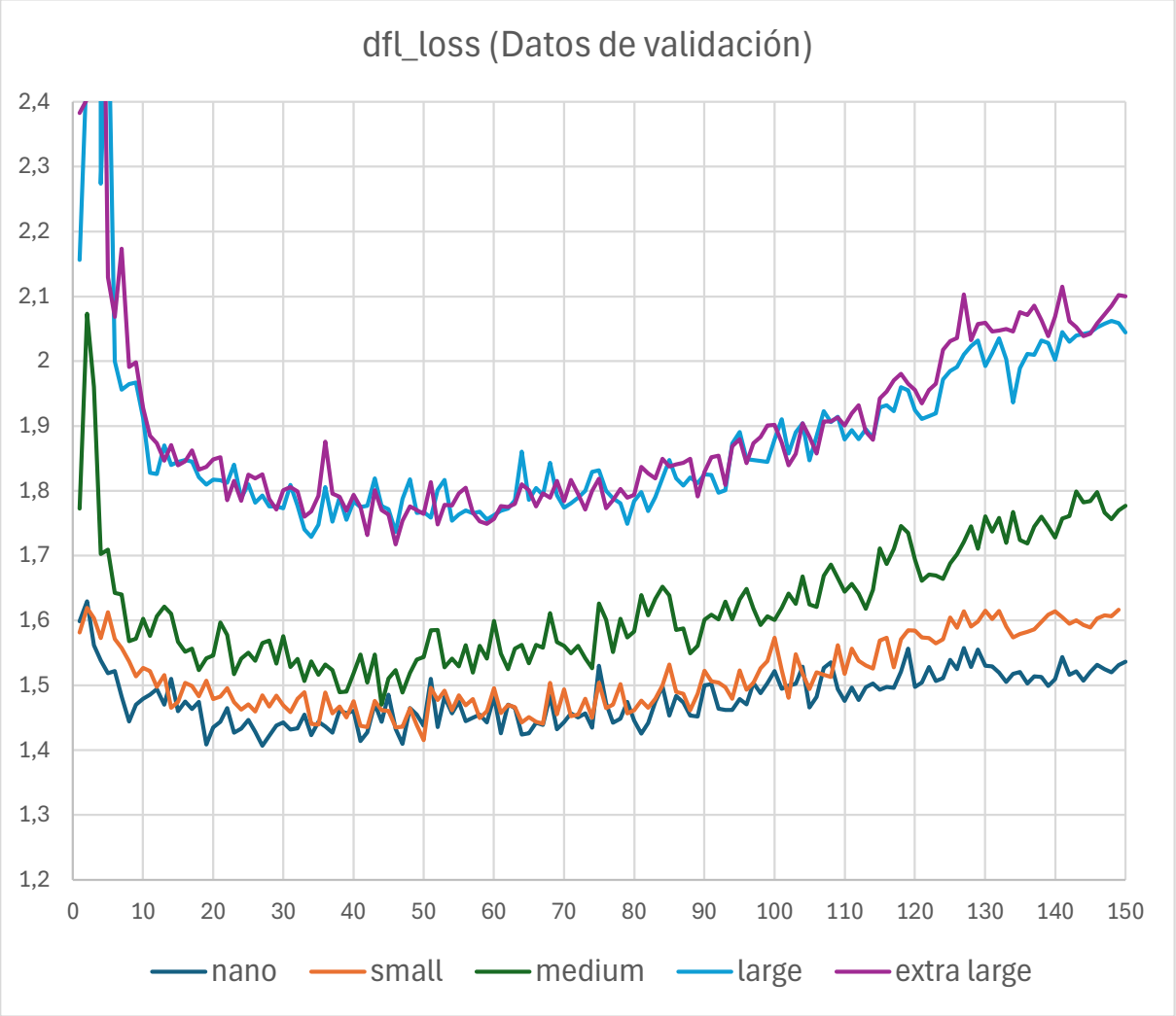



*Figura 44: Evolución de la componente de pérdida dfl_loss para los diferentes modelos*

## 4.2. Influencia del tamaño de lote *(batch).*

A continuación, se analiza el efecto del tamaño del lote, para lo cual se usa el modelo *nano*. Se usan diferentes tamaños de *bath* = 1, 4, 8, 16 y 32. Se deja a cero el parámetro *mosaic* (que combina cuatro imágenes en una sola); es una estrategia de *data augmentation* que usa YOLOv8 por defecto, pero que en los experimentos realizados no se aplica para visualizar mejor el efecto de actualizar los parámetros de la red neuronal con un número de *batch* pequeño.

En teoría, un *batch* alto puede generar sobreajuste, aunque la intuición podría indicar un efecto de regulación es todo lo contrario. Por otra parte, hace que la solución converja con más suavidad, dado que utiliza más datos para actualizar los parámetros (gradientes más suaves) de la red neuronal. Por otro lado, un *batch* bajo introduce más ruido en los gradientes obtenidos de cada *backpropagation*, esta ayuda en teoría a generalizar mejor, pero puede generar problemas de convergencia.

En la gráfica de la Figura 46 se representa la función de pérdida total (7,5box_loss + 0,5cls_loss + 1,5dfl_loss) para los distintos valores de tamaño de lote, tanto para datos de entrenamiento como de validación:

Para tamaño de *batch* pequeño (1-2) presenta un poco más de ruido, especialmente en los datos de validación, lo cual es esperado porque se actualiza el modelo con menos información por paso. Cuando el *batch* es igual a 2 mejora ligeramente ese comportamiento y sobreajusta un poco menos que con *batch* igual a uno. Cabe destacar que con *batch* = 1 generaliza mal, lo cual no cuadra con la teoría, pero puede ser debido a que se procesan pocos datos en cada actualización de parámetros, con valores de *batch* más altos se introduce algo más de aleatoriedad en cada paso. Dado que antes de cada época los datos se barajan (*shuffle*) y luego se generan los bloques para cada paso.

Para tamaños de *batch* intermedios (4-8-16) la pérdida de entrenamiento baja de forma más suave y la pérdida de validación oscila ligeramente menos. Los resultados muestran que los modelos generalizan mejor que *batchs* 1 y 2.

Para el tamaño de *batch* más alto (32) se presenta una peor generalización (valores de la función de pérdida en validación más altos) y la convergencia de la función de pérdida con los datos de entrenamiento no mejora con respecto a los obtenidos con *bacth* 8 y 16. El *batch* 32 es demasiado grande teniendo en cuenta que el *dataset* solo tiene 414 imágenes. Por lo que se determina que un valor de *batch* adecuado para el *dataset* utilizado será de 8 ó 16. Observamos también que a partir de la época 40 todos tienden a sobreajustar, por lo que el modelo deja de generalizar bien, lo cual es normal porque el *dataset* es relativamente pequeño.

En la Figura 45 se representa la función de pérdida total, en línea continua relativa a los datos de entrenamiento y en línea discontinua a los datos de validación. Se observa cómo con *batch* igual a 1 y *batch* igual a 32 generalizan peor que el resto.

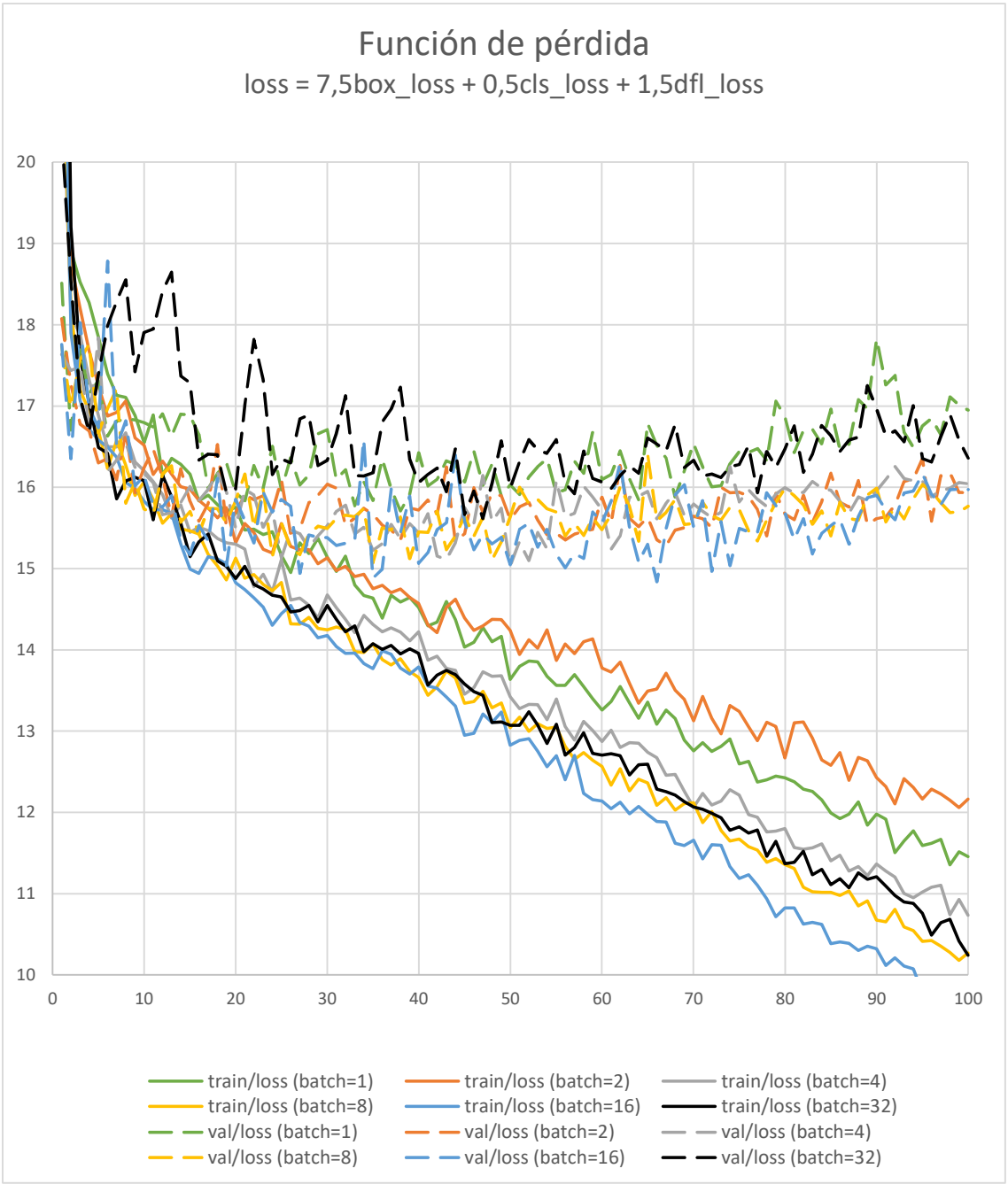



*Figura 45: Función de pérdida para datos de entrenamiento y validación, con diferentes tamaño de lote.*

Se representa a continuación la evolución del *mAP* obtenido para cada tamaño de lote sobre los datos de validación. En general, los valores de *batch* comprendidos entre 4 y 16 dan buenos resultados, generalizando mejor que con el *bath* 32 que presenta muchas oscilaciones.

En general, es interesante utilizar el *batch* más alto posible porque permite mejorar el rendimiento en el entrenamiento, dado que la GPU puede procesar muchos datos en paralelo. Por tanto hay menos pasos de actualización de parámetros por época y esto hace que el entrenamiento sea más rápido. En nuestro caso, usando el modelo nano y con un conjunto de datos tan pequeño, los entrenamientos son bastante rápidos y no será necesario usar *batch* muy alto.

En la Figura 46 se representa la métrica mAP50-95 para diferentes tamaños de *batch*. Observamos que a excepción de *bath* igual a1 y *bath* igual a 32, el resto obtienen un mAP bastante estable durante las épocas de entrenamiento. No hay una mejoría sustancial a partir de la época 40 para ningún tamaño de *batch*.

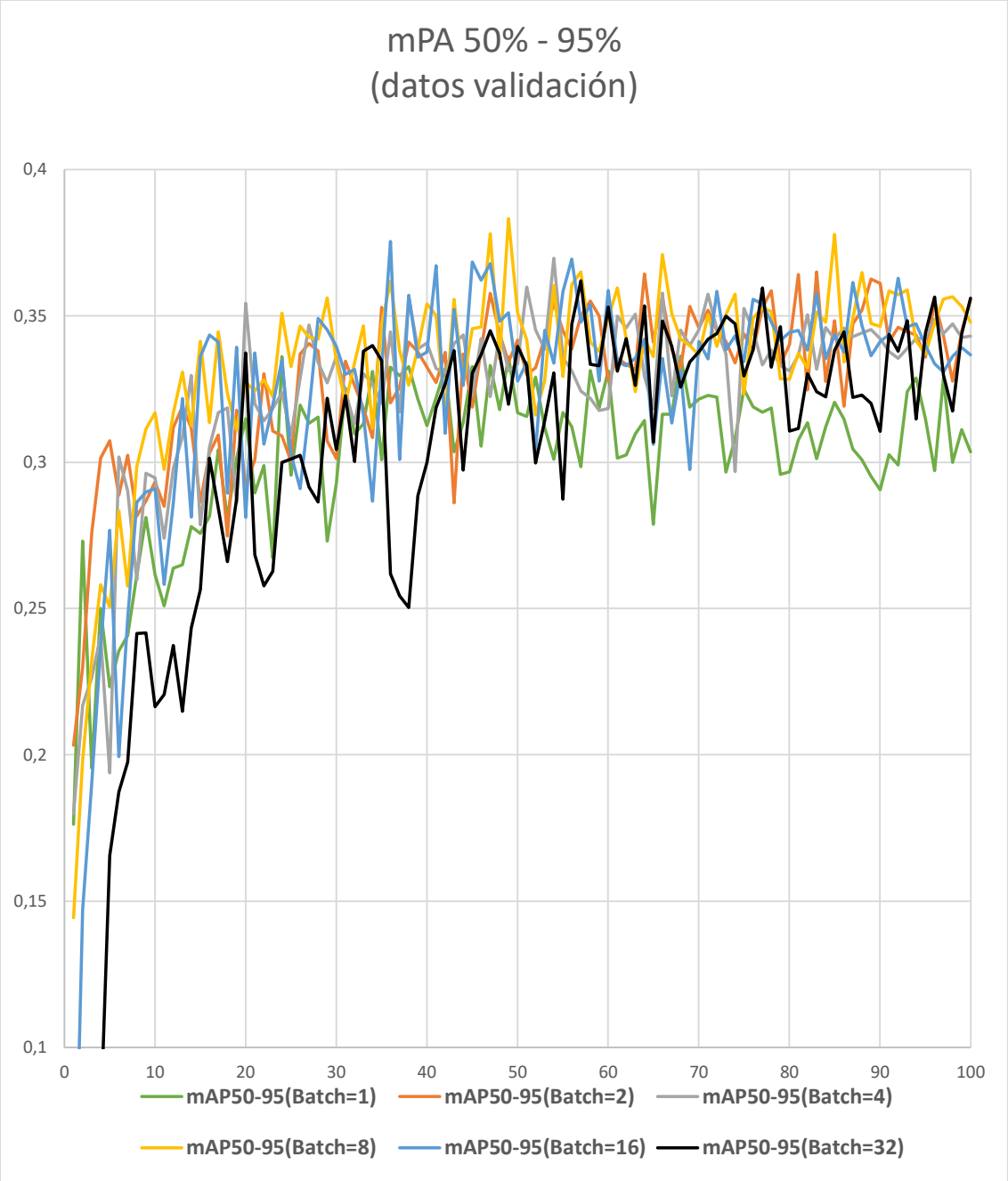



*Figura 46: mAP 50%-95% en datos de validación para diferentes tamaños de lote.*

## 4.3. Influencia de la tasa de aprendizaje (*learning rate).*

A continuación, se analiza el efecto del parámetro *learning rate*. Para lo cual se tienen que determinar una serie de parámetros que utiliza YOLOv8 para ajustar el *learing rate* durante el aprendizaje los cuales se describen la Tabla 8, esta documentación está disponible en https://docs.ultralytics.com/modes/train/#train-settings.

*Tabla 8: Descripción hiperparámetros YOLOv8 relativos al aprendizaje.*

| | | | |
|---|---|---|---|
| **optimizer** | str | 'auto' | Elección del optimizador para la formación. Las opciones incluyen SGD, Adam, AdamW, NAdam, RAdam, RMSProp etc., o auto para la selección automática basada en la configuración del modelo. Afecta a la velocidad de convergencia y a la estabilidad. |
| **lr0** | float | 0.01 | Tasa de aprendizaje inicial (es decir SGD=1E-2, Adam=1E-3). El ajuste de este valor es crucial para el proceso de optimización, ya que influye en la rapidez con la que se actualizan las ponderaciones del modelo. |
| **lrf** | float | 0.01 | Tasa de aprendizaje final como fracción de la tasa inicial = (lr0 * lrf), que se utiliza junto con los programadores para ajustar el ritmo de aprendizaje a lo largo del tiempo. |
| **momentum** | float | 0.937 | Factor de impulso para SGD o beta1 para optimizadores Adam, que influye en la incorporación de gradientes pasados en la actualización actual. |
| **weight_decay** | float | 0.0005 | Término de regularización L2, que penaliza los pesos grandes para evitar el sobreajuste. |
| **warmup_epochs** | float | 3.0 | Número de épocas para el calentamiento de la tasa de aprendizaje, aumentando gradualmente la tasa de aprendizaje desde un valor bajo hasta la tasa de aprendizaje inicial para estabilizar el entrenamiento desde el principio. |
| **warmup_momentum** | float | 0.8 | Impulso inicial para la fase de calentamiento, ajustándose gradualmente al impulso establecido durante el periodo de calentamiento. |
| **warmup_bias_lr** | float | 0.1 | Tasa de aprendizaje de los parámetros de sesgo durante la fase de calentamiento, que ayuda a estabilizar el entrenamiento del modelo en las épocas iniciales. |

Para este análisis interesa dejar el valor del *learning rate* fijo durante todo el proceso de aprendizaje, así se puede valorar mejor su efecto. Para lo cual se ajustan los parámetros tal y como se muestra en la captura de pantalla de la Figura 47. Se ajustan los parámetros para que la tasa de aprendizaje sea constante durante todas las épocas de entrenamiento y se reduce el momento al máximo, dado que el modelo SGD incorpora momento por defecto.

```
# Entrenar el modelo
results = model.train(
    data = path,
    epochs = 100,  # Asegurar que haya un número de épocas definido
    batch = 16,    # Debe ser un entero
    # weight_decay = 0.5,  #como la regularitation L2, pero en actulizacion parametros
    patience = 150,
    # mosaic = 0,
    optimizer ='SGD',
    lr0 = 0.003, # valor de lr que queremos usar
    lrf = 1, # para que no reduzca el lr durante los ulitmos epochs
    momentum = 0.01, # valor muy bajo para tener practicamente SGD sin momento
    warmup_epochs = 0, # sin reduccion en el arranque
    warmup_momentum = 0,
    warmup_bias_lr = 0,
    plots = True,
    device = device  # Usar la detección automática de GPU/CPU
)
```

*Figura 47: Captura pantalla con parte del código del script de entrenamiento.*

Como método de optimización se utiliza SGD (*Stochastic Gradient Descent*), que es la forma más simple de actualizar los parámetros de la red, modificando los parámetros en la dirección del gradiente negativo de la función de pérdida.

$$x += -learning\ rate * dx \qquad (4.1)$$

Donde $x$ representa un vector de parámetros y $dx$ es el gradiente de la función de pérdida respecto de $x$.

A partir del gráfico de la Figura 48 se determina que una tasa de aprendizaje alta de 0.3 genera peores valores de pérdida, esto se debe a que hay demasiada energía en la actualización de los parámetros. Incluso en las primeras épocas la función de pérdida aumenta, lo cual es un síntoma claro de que la tasa de aprendizaje es demasiado alta. En el caso de una tasa de aprendizaje muy baja 0.0001 el aprendizaje se estaca rápidamente y la mejora es más lineal a lo largo de las distintas épocas.

Se incluye también en el gráfico de la Figura 48 la evolución de la función de pérdida con el método AdamW [40] y con tasa de aprendizaje igual a 0.01.

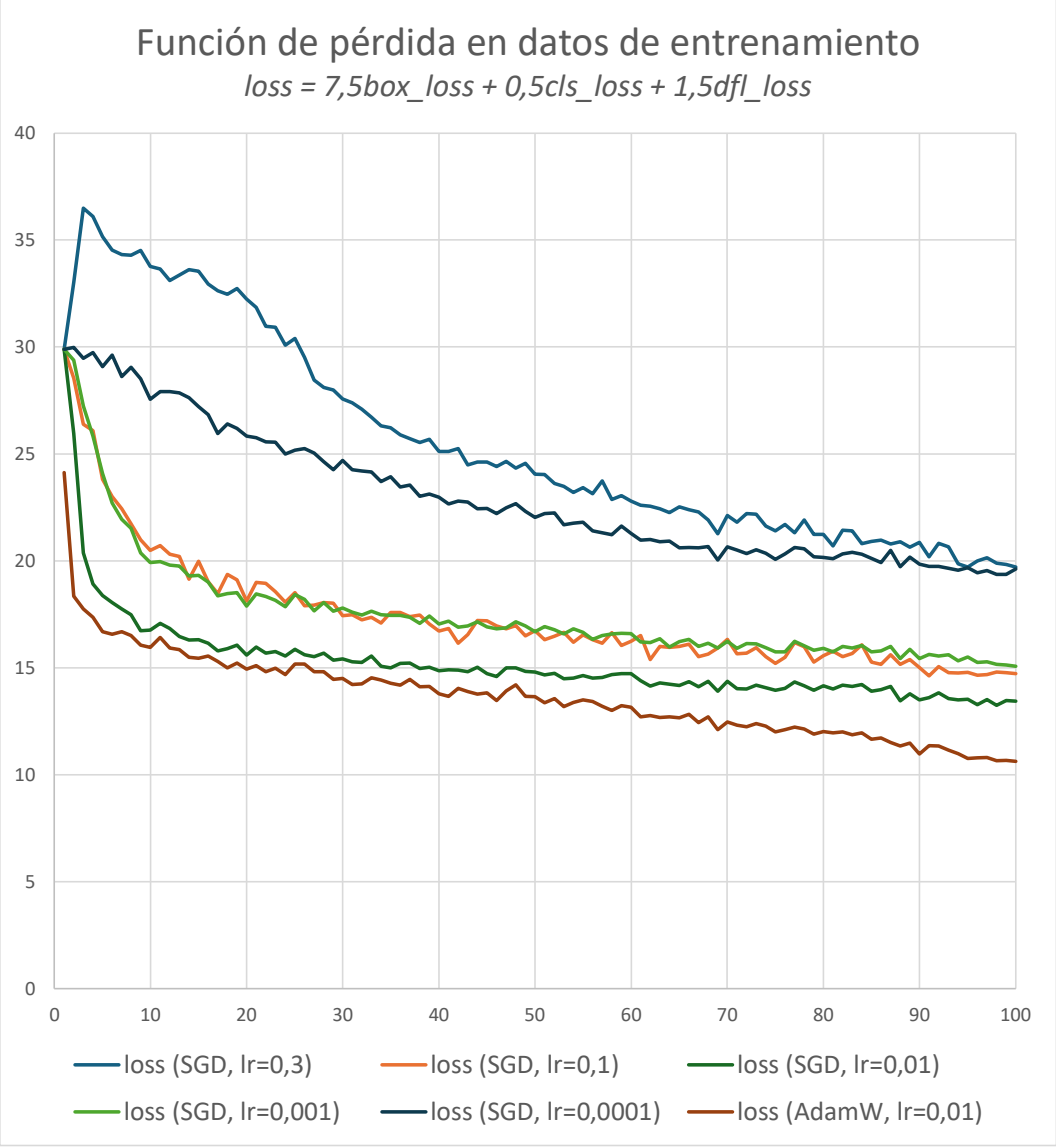



*Figura 48: Función de pérdida durante el entrenamiento para diferentes tasas de aprendizaje.*

El mejor resultado se obtiene con la tasa de aprendizaje de 0.01. No obstante, comparado con el método de optimización por defecto de YOLOV8, esto es AdamW[40] y con la misma tasa de aprendizaje se obtiene una mejor pendiente en la función de pérdida (en datos de entrenamiento).

A partir de los resultados mostrados en la gráfica de la Figura 49, se observa que el comportamiento de la métrica mAP sobre los datos de validación con el optimizador AdamW presenta prácticamente los mismos resultados que con el optimizador SGD.

Se observa también, que incluso utilizando un método tan sencillo como SGD y una tasa de aprendizaje fija, se obtiene una precisión comparable a la lograda con el modelo más complejo AdamW, configurado con los parámetros por defecto de YOLOv8 (*warmup_epoch* = 3 y reducción de la tasa de aprendizaje en las últimas épocas *lrf* = 0.01). Los hiperparámetros *warmup_epoch* y *lrf* se explican en el apartado 4.1.

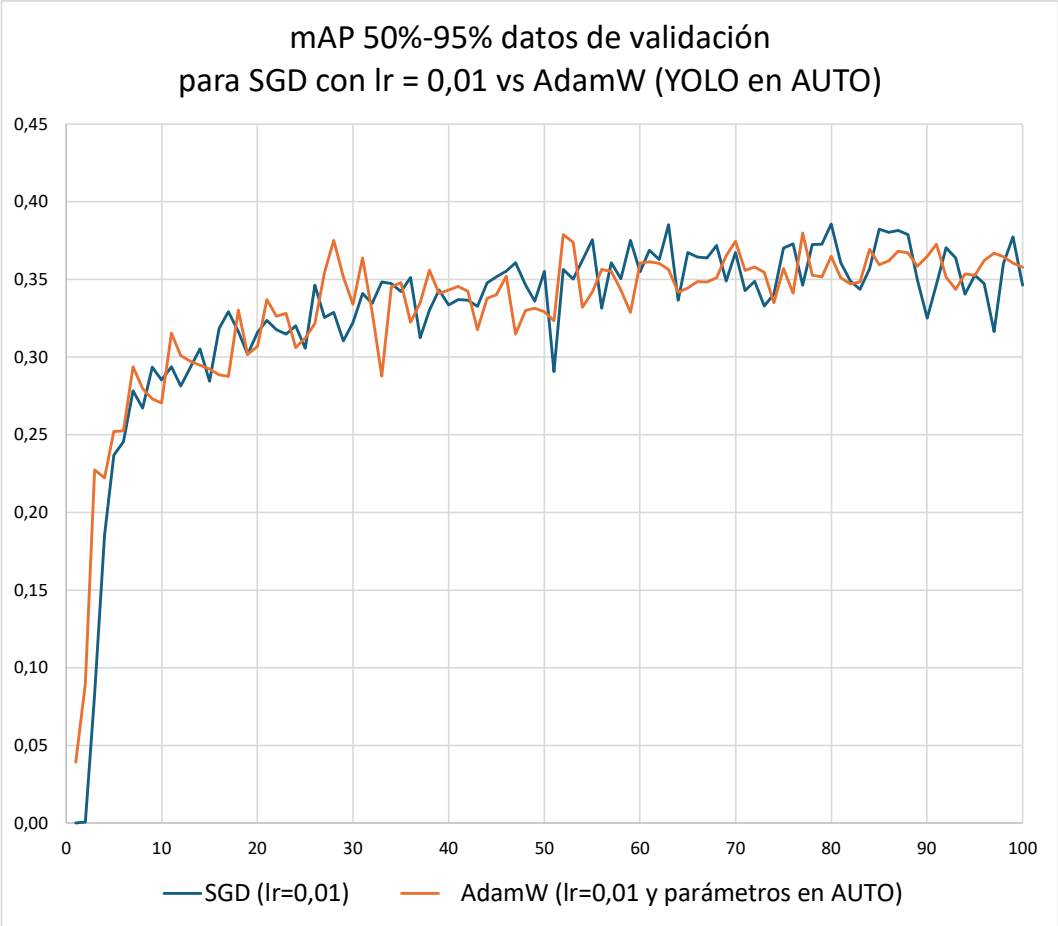



*Figura 49: mAP para optimizador SGD y AdamW[40]. Learning rate=0.01*

La gráfica de la Figura 49 muestra la métrica mAP para dos métodos de optimización, en azul el SGD con un lr=0.01 fijo durante todas las épocas, en naranja el método AdamW con un lr=0.01. Se observa que para nuestro conjunto de datos un modelo tan sencillo con SGD converge bastante bien.

## 4.4. Obtención de la cara frontal de la bobina y posición angular de los flejes radiales.

La intención de este apartado es la de dar una imagen frontal de la cara de bobina que resulte de interés para un inspector de calidad, de tal modo que desde una ubicación en remoto pueda determinar la correcta colocación de los flejes radiales. Por lo que este apartado no tendrá un carácter técnico, sino de exploración de posibilidades prácticas para su aplicación en un entorno industrial y que resulte atractivo a un consumidor de este tipo de soluciones basadas en visión por computador y aplicación de técnicas de *deep learning*.

Para este apartado se usará un modelo de segmentación de instancias de YOLOv8, distinguiendo entre tres clases: cara frontal de la bobina (*face*), agujero de la bobina (*hole*) y parte cilíndrica de la bobina (*coil*).

Para entrenar este modelo es necesario generar mascaras individuales para cada instancia de objeto de la imagen, por lo que el trabajo de generación de datos resulta laborioso.

La Figura 50 muestra una imagen con la predicción generada con modelo yolov8n-seg. Se identifican tres clases: cara de la bobina (*face*), agujero de la bobina (*hole*) y cara cilíndrica (*coil*). Se muestra junto a la clase la confiabilidad de la predicción.

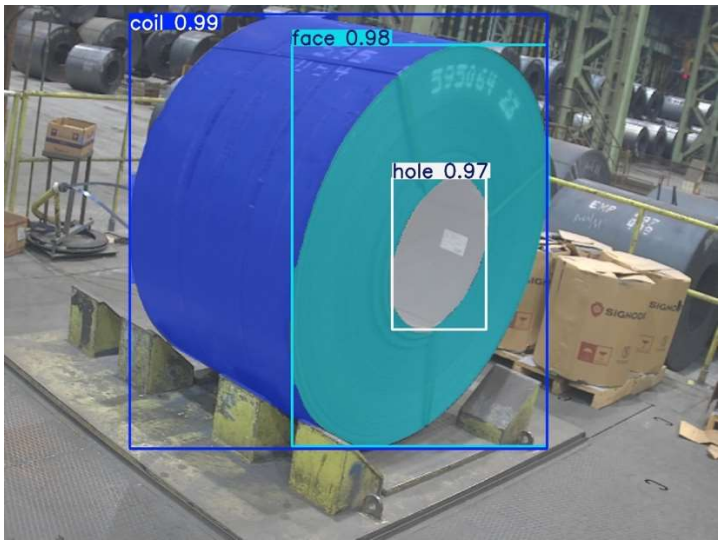



*Figura 50: Predicción modelo yolov8n-seg de segmentación de instancias sobre imagen de test.*

Con la máscara de la cara de bobina, se realiza una selección de esos píxeles y mediante una transformación en perspectiva, usando las librerías de *openCV* se genera una imagen frontal de la cara de la bobina tal y como se representa en la Figura 51. Esta parte de la memoria tiene la intención de explorar de posibilidades de la librería *openCV*.

Para la transformación en perspectiva se identifican 4 puntos de la imagen original y 4 puntos de la imagen resultante. Concretamente se usará la función *cv2.getPerspectiveTransform* para calcular la matriz de transformación entre dos conjuntos de puntos, se necesitan cuatro puntos del origen y cuatro puntos de destino para generar la matriz. Luego la función *cv2.wrapPerspective* toma la imagen de entrada y la transforma según la matriz de perspectiva

proporcionada.

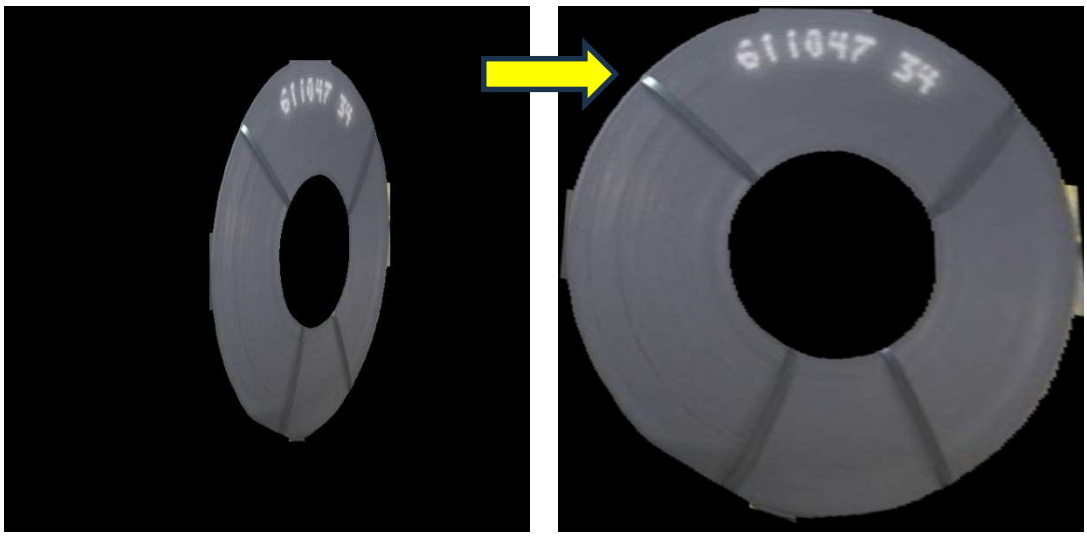


*Figura 51: Transformación en perspectiva de la mascara de la cara de la bobina.*

Con la imagen frontal de la cara de la bobina, se realiza una detección de flejes usando el modelo de detección de objetos, obteniendo la predicción y los *bounding boxes*, tal y como se observa en la imagen de la izquierda de la Figura 52. En la imagen de la derecha se utiliza la posición de esos *bounding boxes* para estimar la posición angular del fleje.

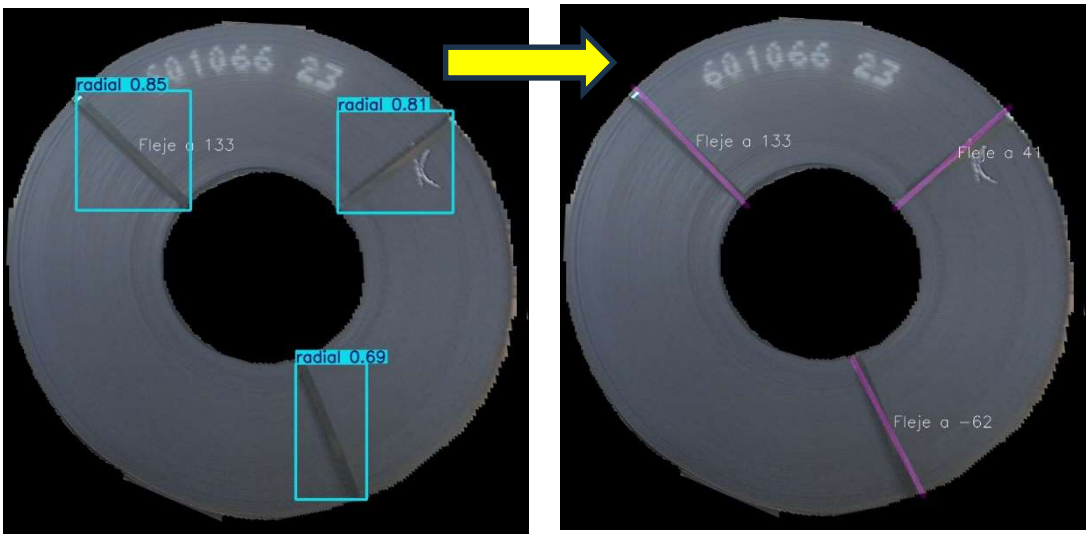



*Figura 52:Predicción posición angular flejes a partir de las cajas delimitadoras.*

Ubicando los *bounding boxes* dentro de los cuatro cuadrantes de la bobina, se determina la posición angular de los flejes a partir de la predicción de los *bounding boxes*, se observa que la predicción del ángulo es suficientemente precisa para supervisar un proceso de flejado manual. No se puede hacer un análisis comparativo, pues no sabemos la posición real de los flejes, dado que solo se dispone de la imagen trasformada. Pero la intención de este apartado es la de exploración de posibilidades para hacer más atractiva a un potencial usuario la solución de visión por computador.

# Capítulo 5

# Conclusiones y trabajo futuro

## 5.1. Conclusiones.

Queda reflejado a lo largo de este trabajo que pueden utilizarse redes neuronales convolucionales para identificación de objetos en entornos industriales sin necesidad de usar modelos excesivamente complejos o necesitar requisitos de hardware excepcionales. En este caso se ha usado la versión 8 de YOLO con el modelo *nano* como más prometedor para la identificación de flejes en bobinas de acero

Es habitual que en las empresas existan circuitos de video y quieran aprovechar esta información para implementar una solución de *Deep Learning*, pero es importante concienciar que, antes de nada, incluso de elegir el método a utilizar, es necesario un *dataset* consistente.

El *Deep Learning* consiste en capacitar redes neuronales profundas para que infieran automáticamente representaciones complejas del conocimiento a partir de datos, aprendiendo patrones y relaciones implícitas sin necesidad de reglas explicitas. Para esto es fundamental contar con conjuntos de datos amplio, representativo y de calidad, ya que el rendimiento del modelo depende de la fiabilidad y riqueza de la información con la que se entrena.

En nuestro caso disponíamos solo de 314 imágenes capturadas desde una cámara fija en una instalación de flejado manual, pero para aumentar la robustez del modelo se añadieron 100 imágenes genéricas de otras instalaciones industriales con un patrón y perspectiva diferentes. La Figura 53 muestra en el conjunto de la izquierda una representación de imágenes capturadas desde una cámara fija de la instalación industrial. En la parte derecha imágenes genéricas de otras instalaciones con diferentes perspectivas y distribución de las bobinas.


*Figura 53: Representación de parte del conjunto de datos.*

Aunque el modelo resultante solo va a tener que inferir sobre bobinas en la estación de flejado, es interesante ampliar el conjunto de datos de tal manera que sea más robusto a pequeñas modificaciones en la orientación de la cámara, cambios de iluminación, presencia de varias bobinas, etc.

Se ha demostrado con el modelo *nano* de YOLOv8 que se puede detectar la presencia de flejes en la bobina. Al tratarse de un modelo muy ligero puede correr incluso en una Raspberry o PC de uso ofimático, no siendo necesario usar equipos con múltiples GPU para la inferencia. No obstante, resulta recomendable usar un equipo potente con GPU para entrenar el modelo.

El nivel de obtención de características del modelo *nano* permite identificar entre 2 clases de fleje con una precisión suficiente, y será más robusto al sobreajuste, al contar con menos parámetros de entrenamiento, esto se infiere de los resultados del apartado 4.1 donde se observa que el modelo nano es menos sensible al sobreajuste en datos de validación. Parece también evidente que para el conjunto de datos del proyecto no tiene sentido usar modelos excesivamente complejos. Pues un modelo con excesivos parámetros tenderá a sobre-ajustar sin mejorar la inferencia en imágenes nuevas.

Otra conclusión importante es que a veces es complicado anotar el *Ground Truth*. En el caso de los flejes circunferenciales, dado su aspecto, es difícil aislarlos en un *Bounding Box* sin incluir elementos anexos, incluso de otra clase. Dado que el etiquetado no es muy preciso no tiene sentido buscar modelos muy exactos en el posicionado.

En la imagen de la Figura 54 se muestra dos opciones para el etiquetado de los flejes circunferenciales, se opta por utilizar la opción de la izquierda, para poder identificar mejor individualmente cada fleje, seleccionando la parte más vertical del fleje. El fleje circunferencia (en amarillo) en la imagen de la derecha, al etiquetarlo, coge parte los tres flejes circunferenciales contiguos y hasta un fleje radial. Para este proyecto se ha utilizado el etiquetado que aparece en la parte izquierda de la imagen.

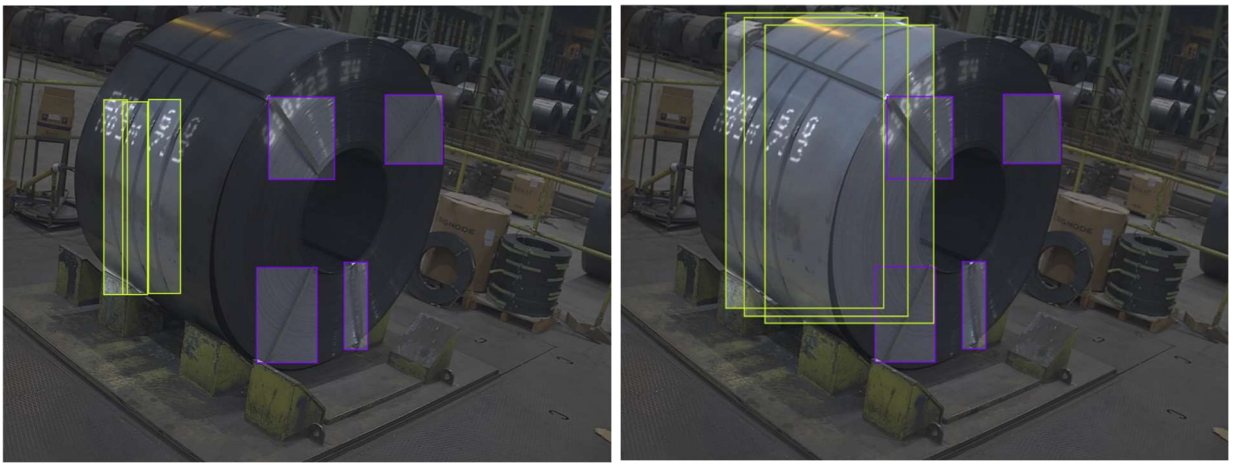


*Figura 54: Posibilidades de etiquetado del Gorund Truth.*

Por tanto, la definición de objetos del *Ground Truth* no siempre es sencilla. En este proyecto la precisión del etiquetado no es muy buena y la naturaleza de los contornos no queda clara.

Respecto a los hiperparámetros, se ha demostrado cómo una correcta tasa de aprendizaje es determinante para una correcta actualización de los parámetros durante el proceso de retro-propagación. Una tasa de aprendizaje muy alta será demasiado energética para buscar un

óptimo y una tasa muy baja no alcanzará el óptimo a pesar de utilizar un número elevado de épocas. Para este caso concreto el modelo más sencillo SGD consistente en actualizar los parámetros con el gradiente negativo de la función de pérdida ha funcionado muy bien, YOLOv8 usa por defecto un modelo mucho más complejo, concretamente AdamW, que para nuestro *dataset* no ha supuesto una mejora destacable.

Respecto al tamaño de *batch*, estábamos condicionado a la memoria de la GPU y al tamaño del *dataset* utilizado, llegando a la conclusión que los mejores valores se han producido para un *batch* de 8 y 16. Valores muy altos generalizan peor y valores muy bajos requieren de más pasos y las actualizaciones de los parámetros en cada fase de retropropagación es más enérgica, en teoría esto debería ayudar a generalizar mejor. En nuestro caso un *batch* de 1 (una sola imagen) y sin usar mosaico daba peores resultados que un *batch* de 2. El resto de los tamaños de *batch* se han comportado según lo esperado.

También ha quedado constancia que con un *dataset* tan pequeño no tiene sentido realizar muchas épocas de entrenamiento, porque a partir de la 30 o 40 el modelo empezaba a sobreajustar, independientemente de los hiperparámetros utilizados. Es decir, con la información disponible no es posible ajustar más el modelo sin sobreajuste. YOLO tiene herramientas para parar el entrenamiento cuando los datos de validación y de entrenamiento empiezan a distanciarse, para el proyecto se deshabilitó esta opción precisamente para corroborar el sobreajuste.

## 5.2. Trabajo futuro

Como continuación a este trabajo se podrían realizar varias pruebas, que se detallan a continuación.

La primera opción podría consistir en probar con otro etiquetado, como el mostrado en la imagen derecha de la Figura 54, con un *Bounding Box* más amplio que abarque toda la extensión del fleje circunferencial aunque incluya objetos contiguos. Sin embargo, lo más interesante sería diseñar un modelo de segmentación de instancias mediante máscaras de clase, aunque esto requiere un esfuerzo considerable en la etapa de etiquetado, que podría ocupar varias jornadas de trabajo. Aplicaciones como Roboflow tienen incorporados algoritmos inteligentes que infieren de manera bastante precisa las clases y ayudan a reducir parte del trabajo necesario para generar las máscaras de cada instancia; aun así, el tiempo de etiquetado de cada imagen sigue siendo mucho mayor que el del etiquetado de objetos mediante cajas delimitadoras. De hecho, esta opción fue descartada para esta proyecto debido al elevado tiempo necesario para generar el conjunto de datos destinados a entrenar un modelo de segmentación de instancias.

Respecto a la posición angular de los flejes radiales, se podría aplicar técnicas clásicas de visión por computador como la utilización de la transformada de Hough de detección de líneas, que nos permitiría identificar los flejes como líneas y determinar su posición angular.

# Bibliografía

[1] G. Jocher, A. Chaurasia, and J. Qiu, "YOLO by Ultralytics," GitHub, 2024. Disponible on-line: https://github.com/ultralytics/ultralytics. Accedido: 25-agos-2024.

[2] Ultralytics, "YOLOv8—Ultralytics YOLOv8 Documentation," 2024. Disponible on-line: https://docs.ultralytics.com/models/yolov8/. Accedido: 25-agos-2024.

[3] W. Liu et al., "SSD: Single shot multibox detector," in Proc. Eur. Conf. Comput. Vis. (ECCV), Amsterdam, The Netherlands, Oct. 2016, pp. 21–37.

[4] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), Columbus, OH, USA, Jun. 2014, pp. 580–587.

[5] R. Girshick, "Fast R-CNN," en Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 1440–1448.

[6] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," IEEE Trans. Pattern Anal. Mach. Intell., vol. 38, no. 6, pp. 1137–1149, Jun. 2016.

[7] Ö. Can, "CircleDetector Dataset, Open Source Dataset," 2025. License: CC BY 4.0. Disponible on-line: https://universe.roboflow.com/omer-can/circledetector, Marzo 2022. Accedido: Mar. 2025.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," Advances in Neural Information Processing Systems, vol. 25, pp. 1090–1098, 2012.

[10] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.

[11] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, and A. Rabinovich, "Going deeper with convolutions," en Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2015, pp. 1–9.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," en Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2016, pp. 770–778.

[13] J. Dai, Y. Li, K. He, and J. Sun, "R-FCN: Object detection via region-based fully convolutional networks," en Advances in Neural Information Processing Systems, 2016, pp. 379–387.

[14] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," en Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 2980–2988.

[15] T. Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," en Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 2980–2988.

[16] J. R. Terven and D. M. Córdova-Esparza, "A comprehensive review of YOLO architectures in computer vision: from YOLOv1 to YOLOv8," arXiv, 2023. Disponible on-line: https://arxiv.org/html/2304.00501v6.

[17] Y. Wu and K. He, "Group normalization," en Proceedings of the European Conference on Computer Vision (ECCV), 2018. Disponible on-line: https://arxiv.org/pdf/1803.08494.

[18] Y. Wu and K. He, "Group normalization," en Proceedings of the European Conference on Computer Vision (ECCV), 2018. Disponible on-line: https://arxiv.org/pdf/1803.08494.

[19] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, vol. 15, no. 1, pp. 1929–1958, 2014.

[20] A. Zhang, Z. Lipton, M. Li, and A. J. Smola, "8.6. Residual Networks (ResNet) and ResNeXt," en Dive into Deep Learning, Cambridge University Press, 2024, ISBN 978-1-009-38943-3.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," arXiv:1603.05027, 2015.

[22] A. Canziani, E. Culurciello, and A. Paszke, "An analysis of deep neural network models for practical applications," arXiv, 14 Apr. 2017. Disponible on-line: https://arxiv.org/pdf/1605.07678v4.

[23] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), 2016, pp. 779–788. https://doi.org/10.1109/CVPR.2016.91.

[24] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes (VOC) challenge," Int. J. Comput. Vis., vol. 88, no. 2, pp. 303–338, 2010. https://doi.org/10.1007/s11263-009-0275-4.

[25] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," en Proc. Eur. Conf. Comput. Vis. (ECCV), 2014, pp. 740–755, Springer. https://doi.org/10.1007/978-3-319-10602-1_48.

[26] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," arXiv, 25 Dec. 2016. Disponible on-line: https://arxiv.org/abs/1612.08242.

[27] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," arXiv, 8 Apr. 2018. Disponible on-line: https://arxiv.org/abs/1804.02767.

[28] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal speed and accuracy of object detection," arXiv, 23 Apr. 2020. Disponible on-line: https://arxiv.org/abs/2004.10934.

[29] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," en Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 936–944.

[30] G. Ghiasi, T.-Y. Lin, and Q. V. Le, "Dropblock: A regularization method for convolutional networks," Advances in Neural Information Processing Systems, vol. 31, 2018.

[31] C.-Y. Wang, H.-Y. M. Liao, Y.-H. Wu, P.-Y. Chen, J.-W. Hsieh, and I.-H. Yeh, "CSPNet: A new backbone that can enhance learning capability of CNN," en Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, 2020, pp. 390–391.

[32] D. Misra, "Mish: A self-regularized non-monotonic neural activation function," arXiv preprint arXiv:1908.08681, vol. 4, no. 2, pp. 10–48550, 2019.

[33] Ultralytics, "YOLOv5: A family of object detection architectures and models pretrained on the COCO dataset," repositorio GitHub, 2020. Disponible on-line: https://github.com/ultralytics/yolov5.

[34] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," Adv. Neural Inf. Process. Syst., vol. 32, pp. 8024–8035, 2019.

[35] C. Li, Z. Zhang, J. Ma, G. Wu, H. Liu, C. Qian, et al., "YOLOv6: A single-stage object detection framework for industrial applications," arXiv preprint arXiv:2209.02976, 2022. Disponible on-line: https://arxiv.org/abs/2209.02976.

[36] C. Y. Wang, A. Bochkovskiy, and H. Y. M. Liao, "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," arXiv preprint arXiv:2207.02696, 2022. Disponible on-line: https://arxiv.org/abs/2207.02696.

[37] Ultralytics, "YOLOv8: A family of object detection architectures and models pretrained on the COCO dataset," GitHub repository, 2024. Disponible on-line: https://github.com/ultralytics/yolov8.

[38] Z. Zheng, P. Wang, W. Liu, J. Li, R. Ye, and D. Ren, "Distance-IoU loss: Faster and better learning for bounding box regression," Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, no. 07, pp. 12993–13000, 2020.

[39] Z. Li, K. Wang, W. Zhang, J. Sun, and Y. Jiang, "Generalized focal loss V2: Learning reliable localization quality estimation for dense object detection," Advances in Neural Information Processing Systems, vol. 35, pp. 2551–2564, 2022.

[40] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," en International Conference on Learning Representations (ICLR), 2019. Disponible on-line: https://arxiv.org/abs/1711.05101.

[41] Ultralytics, "YOLOv5: A state-of-the-art object detection model," GitHub repository, 2020. Disponible on-line: https://github.com/ultralytics/yolov5.

[42] Ultralytics, "Data augmentation techniques in YOLOv5," Ultralytics, 2024. Disponible on-line: https://docs.ultralytics.com/es/yolov5/tutorials/architecture_description/#2-data-augmentation-techniques.

[43] Ultralytics, "YOLOv5 architecture description," Ultralytics, 2024. Disponible on-line: https://docs.ultralytics.com/es/yolov5/tutorials/architecture_description/.

[44] Ultralytics, "YOLOv8: A state-of-the-art object detection model," GitHub repository, 2023. Disponible on-line: https://github.com/ultralytics/ultralytics.

[45] OpenMMLab, "YOLOv8 graphical representation," GitHub repository, 2024. Disponible on-line: https://github.com/open-mmlab/mmyolo/tree/main/configs/yolov8.

[46] Ultralytics, "YOLOv8 performance metrics," Ultralytics, 2024. Disponible on-line: https://docs.ultralytics.com/models/yolov8/#performance-metrics.