



## Visión por Computadora II - CEAI - FIUBA



Profesores:

- Rafel Alfonso - [alfonsorafel93@gmail.com](mailto:alfonsorafel93@gmail.com)
- Cornet Juan Ignacio - [juanignaciocornet@gmail.com](mailto:juanignaciocornet@gmail.com)
- Seyed Pakdaman - [khodadad.pakdaman@gmail.com](mailto:khodadad.pakdaman@gmail.com)

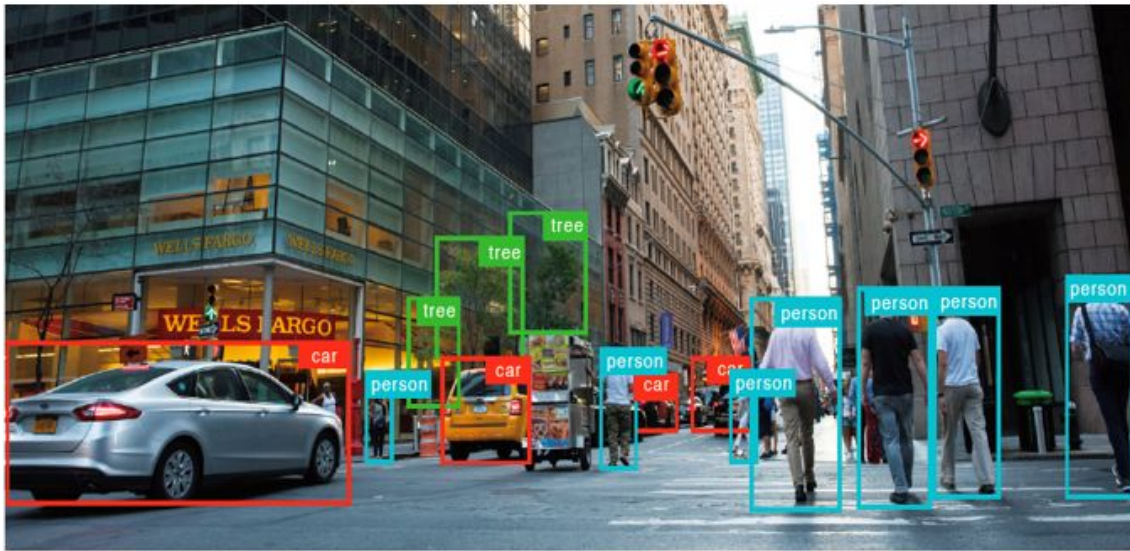
# Tercer clase:



- Problema detección de objetos
- Algoritmo Sliding Windows
- Métrica mAP
- Arquitecturas de 2 etapas:
  - R-CNN
  - Fast R-CNN
  - Faster R-CNN
- Arquitecturas de 1 etapa:
  - YOLO
  - SSD
  - RetinaNet
  - CornerNet
- Etiquetado
- Entrenamiento con Roboflow

# Detección de objetos (Object detection)

Es una combinación de clasificación y localización. Existen métodos de dos etapas (R-CNN, Fast R-CNN, Faster R-CNN), que tienen mejor precisión, y métodos de una etapa (YOLO, SSD, RetinaNet) que tienen mejor velocidad de inferencia.



Datasets: [COCO](#), [Pascal VOC](#), Imagenet, Open Images, etc

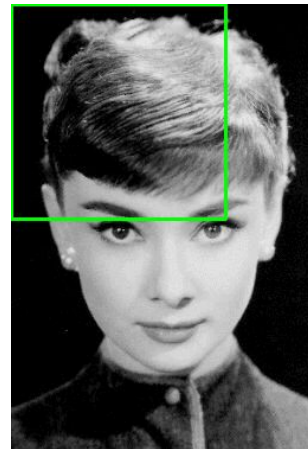
# Algoritmo Sliding Window

Una primera forma de encarar el problema de **detección de objetos** sería intentar utilizar alguna de las redes de clasificación de imágenes ya conocidas pasándole como entrada ventanas de la imagen sobre la cual queremos detectar los objetos.

Tendríamos un **mapa de probabilidades de cada clase** sobre el cual podemos inferir la ubicación de un objeto.

## *Desventajas*

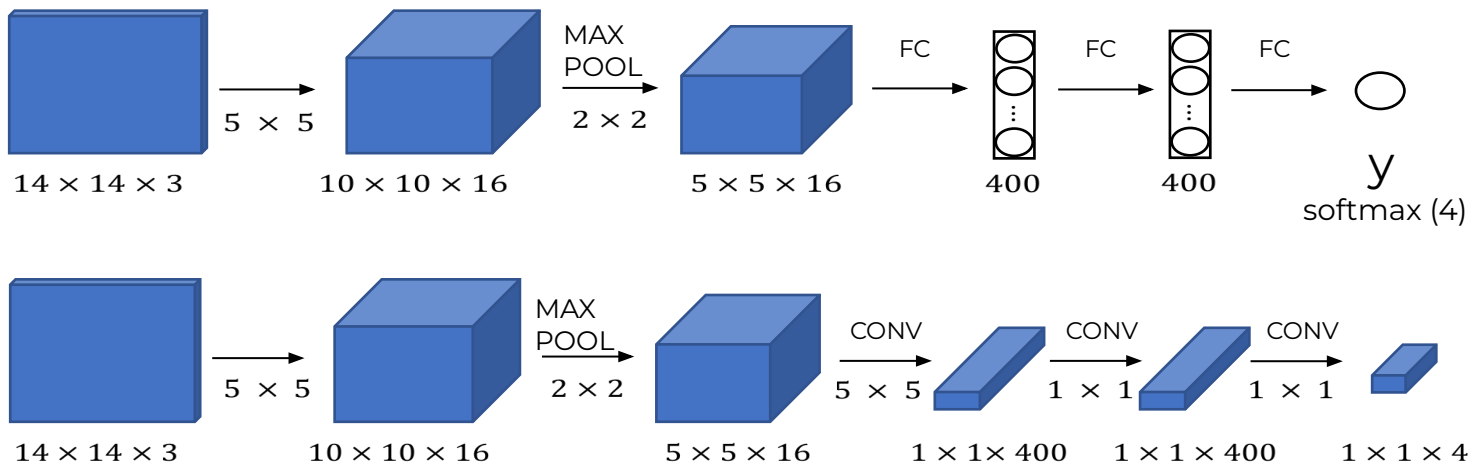
- Esta forma de obtener la ubicación de un objeto **resulta bastante costosa computacionalmente** debido a la cantidad de veces que hay que procesar imágenes con la red convolucional.
- Si los objetos tienen distintos tamaños, **lo ideal sería realizar pasadas con diferentes tamaños de ventana**, lo cual incrementa aún más dicho costo.



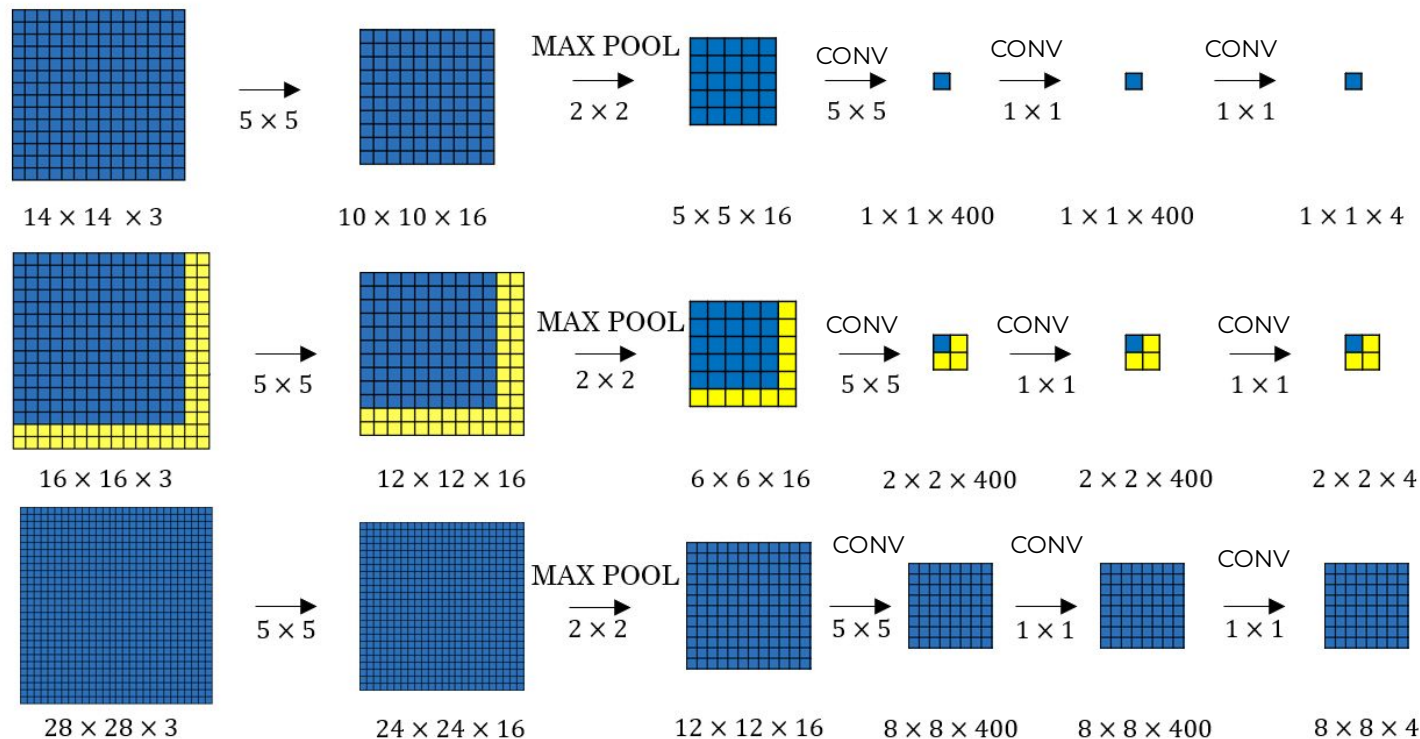
# Algoritmo Sliding Window Convolutacional

Para solventar los problemas de costo computacional, se puede implementar el algoritmo de Sliding Window de forma convolutacional, lo cual implica que debemos transformar las capas densas de nuestra red convolutacional en capas convolucionales.

Supongamos este caso de una red que toma imágenes de  $14 \times 14$  pixeles y predice 4 clases:



# Algoritmo Sliding Window Convolutacional



# mAP: Mean Average Precision



A la hora de medir la performance de los distintos algoritmos de detección de objetos no solo debemos tener en cuenta que la **clasificación del objeto** corresponda a la clase correcta, sino que también, la **ubicación del bounding box** predicho sea acorde al tamaño y ubicación real del objeto detectado.

Para evaluar ambas predicciones a la vez se utiliza el **Average Precision (AP)**, el cual puede obtenerse como el área bajo la curva de Precision-Recall para cada una de las clases. Luego, al promediar los valores de AP entre todas las clases, se obtiene lo que se conoce como **Mean Average Precision (mAP)**, que es la métrica más popular entre las utilizadas para evaluar dichos modelos.

Aun así, cada competencia definió distintas variantes para calcular el valor final de mAP. En este caso analizaremos el propuesto originalmente en la competencia de PASCAL VOC.

# mAP: Mean Average Precision

Por otro lado, en un problema de detección de objetos, ¿de qué forma se define si una predicción es correcta o no?

A diferencia del caso de clasificación, en detección se deben cumplir tres requisitos:

- Que la confianza  $p$  de que existe un objeto sea mayor que un cierto umbral predefinido. En ocasiones esta confianza se expresa como una clase mas representando el fondo.
- Que la clase predicha sea la correcta.
- Que la IoU del bounding box predicho y el ground truth sea mayor que un cierto umbral predefinido.

$$y = \begin{bmatrix} p \\ x \\ y \\ w \\ h \\ c_1 \\ c_2 \end{bmatrix}$$

$p$  es el nivel de confianza de que exista un objeto. Si no hay objeto en la imagen, la etiqueta contiene un 0 en la posición de  $p$ .

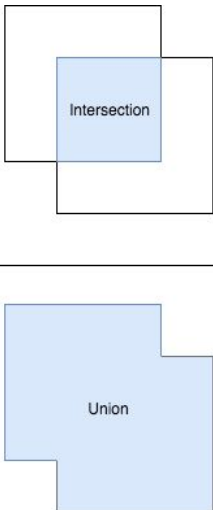
$x, y, w$  y  $h$  definen el bounding box y se utilizan para computar el IoU.

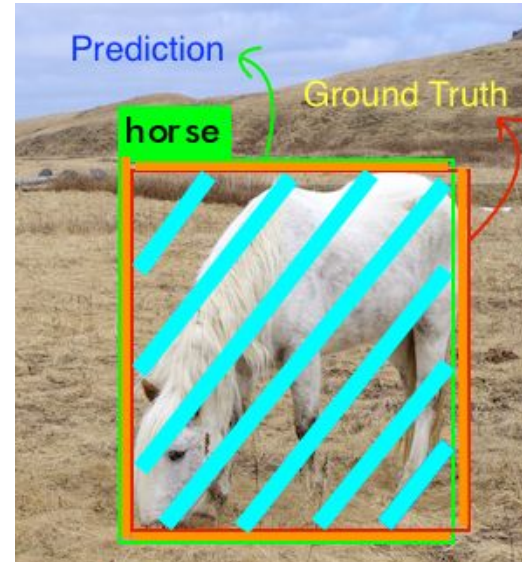
Los valores de  $c$  son las probabilidades de que el objeto corresponda a alguna de las clases del dataset.



# mAP: Intersection over Union (IoU)

También llamada Jaccard Index, es una forma de medir que tan bien coincide la bounding box predicha con la verdadera. Cuanto más cercano a 1 es el valor, mejor es la predicción de dicha bounding box.

$$\text{IoU} = \frac{\text{Intersection}}{\text{Union}}$$




# mAP: Mean Average Precision



Para obtener la curva Precision-Recall, debemos poder calcular ambas métricas por separado. Podemos hacer esto dado que estamos trabajando sobre un conjunto de datos del cual conocemos, previamente, todas sus etiquetas.

Para refrescar..

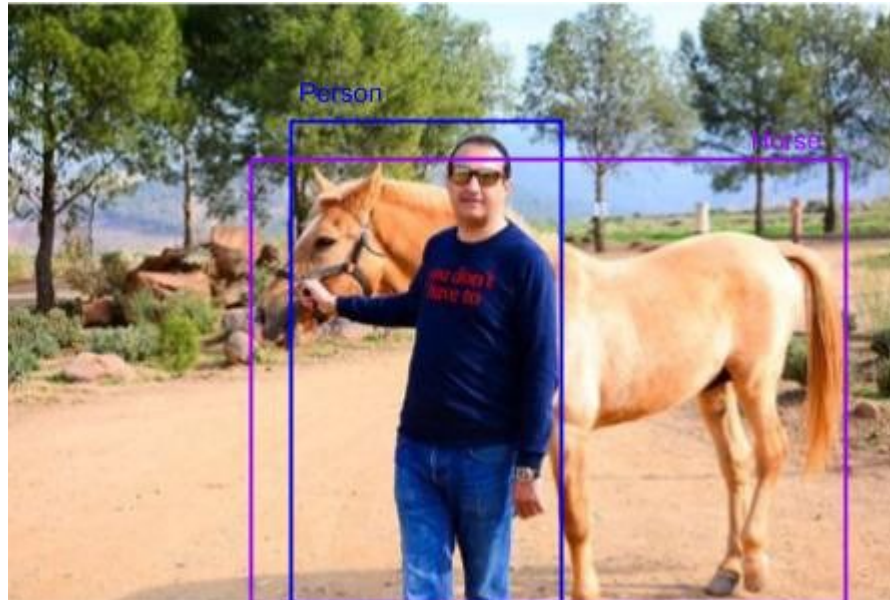
- Precisión responde a la pregunta: ¿Cuando el modelo predice, con que frecuencia lo hace correctamente?
- Recall responde a la pregunta: ¿Con qué frecuencia predijo el modelo correctamente cada vez que debería haberlo hecho?

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall (TPR)} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

# mAP: Mean Average Precision

Supongamos que tenemos la siguiente imagen con dos etiquetas:



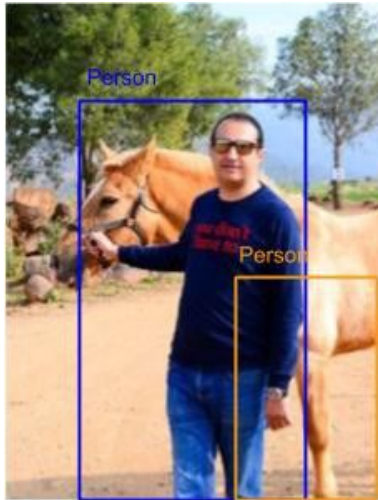
# mAP: Mean Average Precision

Definimos, entonces, que una predicción se considera **TP** si la confianza en que existe un objeto es mayor a cierto umbral, si la clase persona está bien predicha y si el IoU es mayor a cierto valor.



# mAP: Mean Average Precision

La predicción será **FP** si, por ejemplo, el IoU no supera el umbral establecido, si hay BB duplicados (en ese caso solo se toma uno como correcto) o si directamente no hay intersecciones entre el BB y el ground truth.



IoU < 0.5



Duplicate BB are considered as FP



No IoU

# mAP: Mean Average Precision

Por último, la predicción será **FN** si el modelo no es capaz de realizar la detección o si la clase predicha no es la correcta.





# mAP: Mean Average Precision

Veamos cómo se obtiene el mAP con un ejemplo. Supongamos que tenemos un modelo que está entrenado para detectar gatos. Además, tenemos un conjunto de 12 imágenes de test, con 12 etiquetas y recibimos del modelo, 12 predicciones.



# mAP: Mean Average Precision

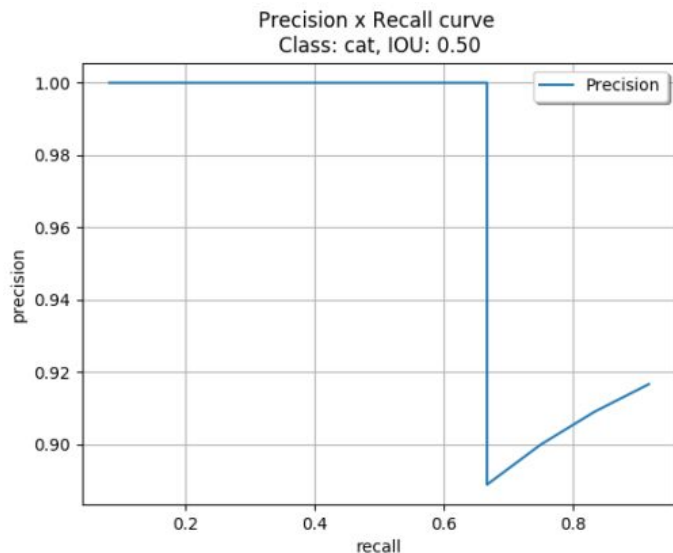
Primero ordenamos las predicciones por nivel de confianza y evaluamos si pasan el umbral de IoU, que, en este caso, lo dejamos en 0.5. Luego, vamos obteniendo los valores de precision y recall (últimas dos columnas de la tabla)

bounding box	confidence ( $\tau$ )	IOU	IOU > 0.5?	$\sum TP(\tau)$	$\sum FP(\tau)$	$Pr(\tau)$	$Rc(\tau)$
D	99%	0.91	Yes	1	0	1.0000	0.0833
K	98%	0.70	Yes	2	0	1.0000	0.1667
C	95%	0.86	Yes	3	0	1.0000	0.2500
H	95%	0.72	Yes	4	0	1.0000	0.3333
L	94%	0.91	Yes	5	0	1.0000	0.4167
I	92%	0.86	Yes	6	0	1.0000	0.5000
A	89%	0.92	Yes	7	0	1.0000	0.5833
F	86%	0.87	Yes	8	0	1.0000	0.6667
J	85%	-	No	8	1	0.8889	0.6667
B	82%	0.84	Yes	9	1	0.9000	0.7500
E	81%	0.74	Yes	10	1	0.9091	0.8333
G	76%	0.76	Yes	11	1	0.9167	0.9167



# mAP: Mean Average Precision

Una vez obtenidos los valores de precision y recall para el conjunto de datos, los llevamos a una gráfica.

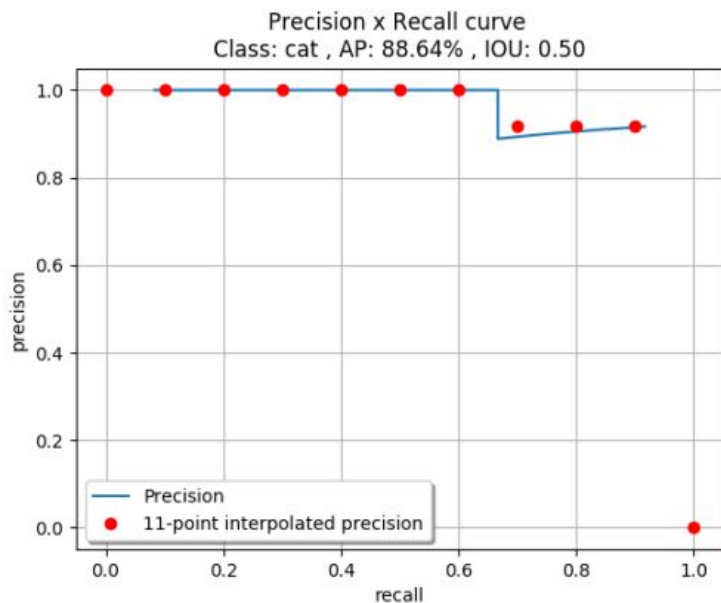


Notar que:

- Para diferentes valores de umbral de IoU, la gráfica puede tomar formas distintas.
- Independientemente del valor de IoU que se elija, el recall nunca llegará a 100 porque hubo un FN.

# mAP: Mean Average Precision

El valor final de la métrica de AP puede obtenerse de distintas formas. En este caso se interpola la gráfica tomando N=11 puntos sobre la misma para calcular el valor final.



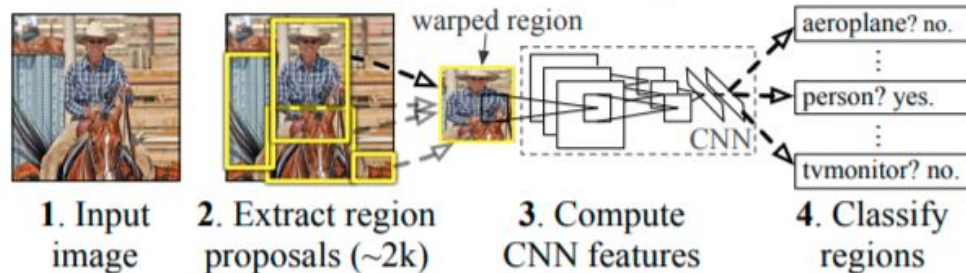
En este caso el cálculo queda:

$$AP = (7 \times 1.0 + 3 \times 0.9 + 1 \times 0.0) / 11 = 0.88$$

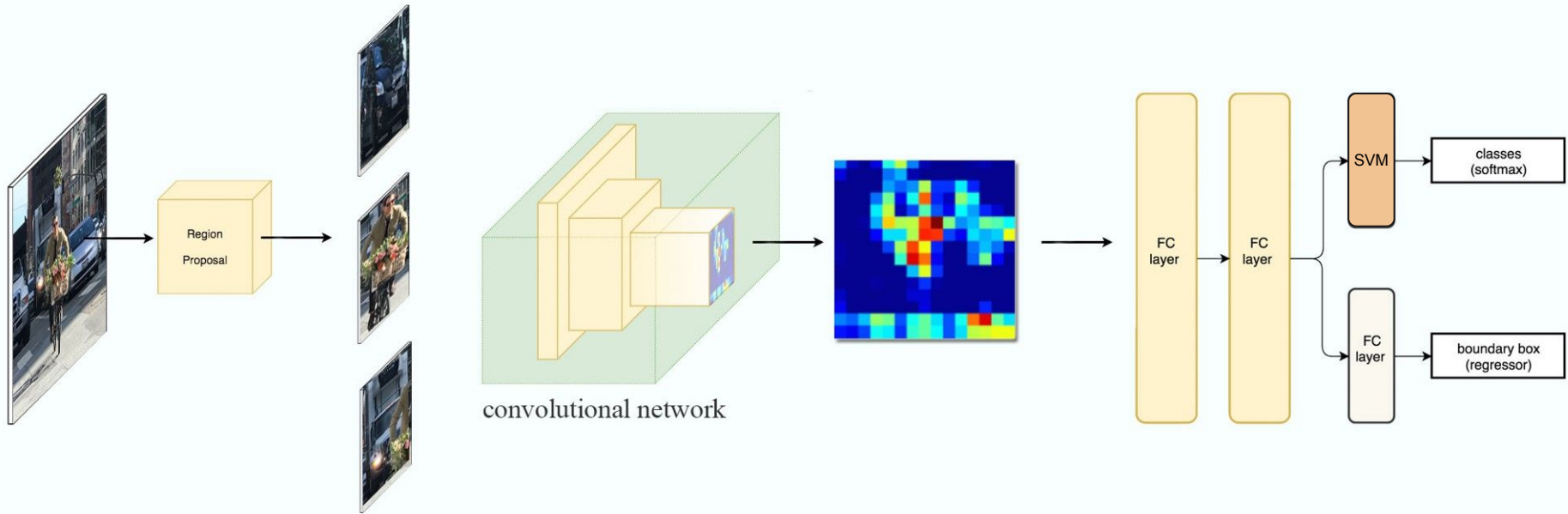
# Arquitecturas clásicas: R-CNN

En 2014 se publicó el primero de una serie de papers sobre algoritmos de detección de objetos, los cuales mejoraron sustancialmente las métricas obtenidas hasta ese entonces en este tipo de problema. El primero de estos propone una arquitectura conocida como R-CNN, basada en una red convolucional. A este, y a sus predecesores, se los considera metodos de dos etapas:

1. En una primer etapa se generan regiones de interés o Region Proposals (ROIs)
2. En la segunda etapa se realiza la clasificación y localización sobre dichas regiones.



# Esquema de R-CNN



What do we learn from region based object detectors (Faster R-CNN, R-FCN, FPN)? [Link](#)

# Arquitecturas clásicas: R-CNN



A pesar de haber sido una mejora importante para su época, la arquitectura de R-CNN presenta varias falencias:

- **No es entrenable de forma end-to-end**, es decir, es necesario realizar varias etapas de entrenamiento distintas que no son paralelizables.
- **Tiene tiempos de inferencia demasiado grandes** para considerarlo viable para una aplicación en tiempo real, fundamentalmente debido a tener que procesar muchas regiones donde en realidad no hay ningún objeto, de forma secuencial.

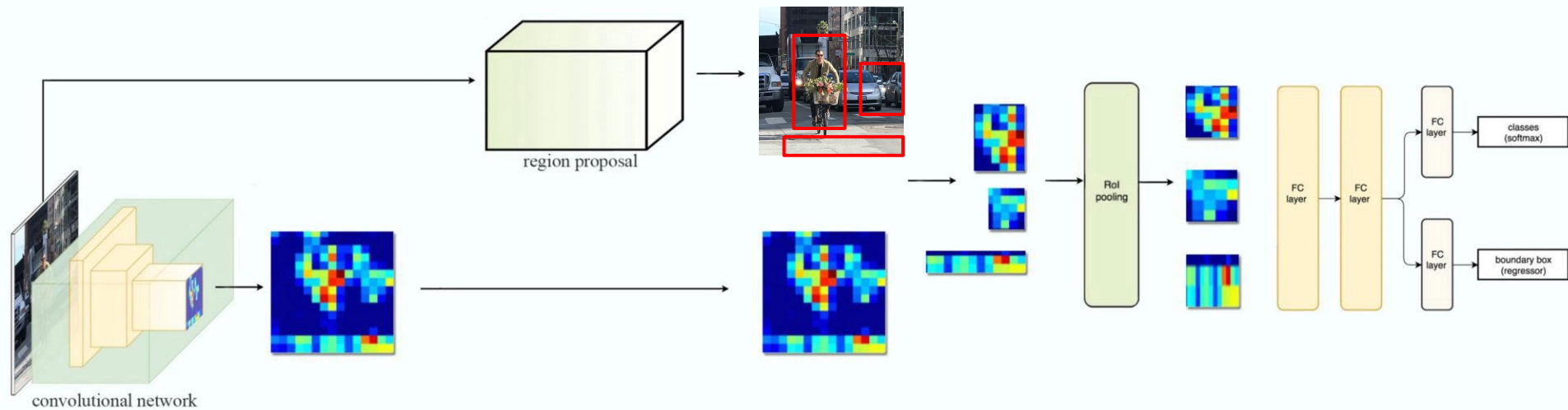
# Arquitecturas clásicas: Fast R-CNN



En 2015 se propuso una mejora al modelo original, la cual se llamó Fast R-CNN. En esta arquitectura se hizo foco, fundamentalmente, en **reducir los tiempos de entrenamiento e inferencia del modelo**.

Esto se logró gracias a que se paralelizó el proceso de extracción de features de la imagen y le eliminaron los SVM para la clasificación de los objetos, dejando todas las partes con base en alguna red neuronal, lo que permitió que todo el sistema pueda ser entrenado al mismo tiempo.

# Esquema de Fast R-CNN



What do we learn from region based object detectors (Faster R-CNN, R-FCN, FPN)? [Link](#)

# Arquitecturas clásicas: Faster R-CNN

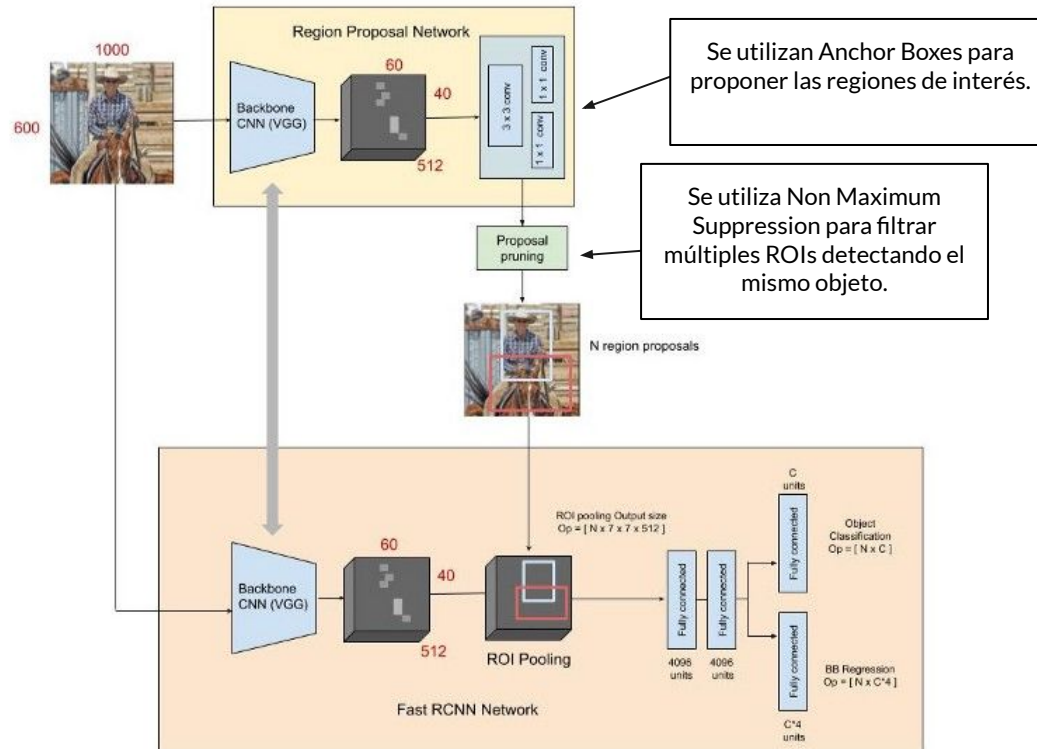


Para cerrar esta trilogía, en 2016 se publicó el paper de Faster R-CNN, una arquitectura muy similar a su predecesora cuya **principal diferencia radica en la forma en la que se obtienen las regiones de interés dentro de la imagen.**

Hasta ese entonces, se venía utilizando la búsqueda selectiva para obtener dichas ROIs. Si bien resulta un método simple y efectivo, es un algoritmo que corre en CPU y es lento. Por ejemplo, en Fast R-CNN, para una predicción que toma 2.3 segundos, el procesamiento para obtener las ROIs lleva 2 segundos.



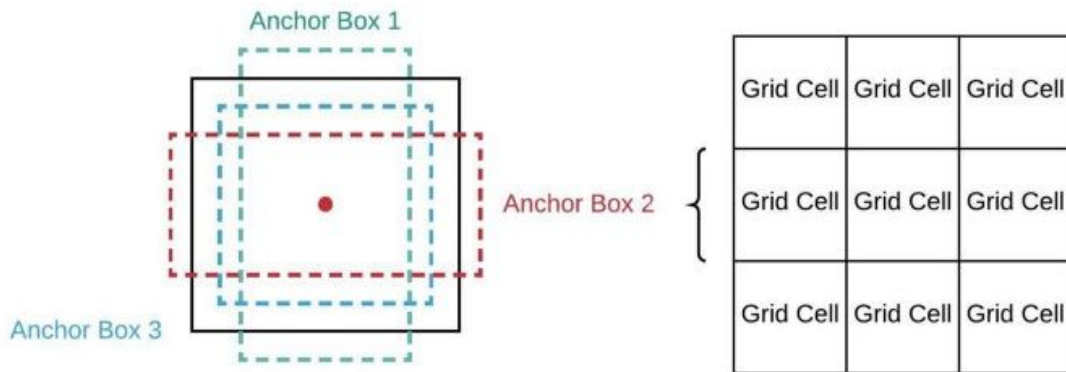
# Esquema de Faster R-CNN



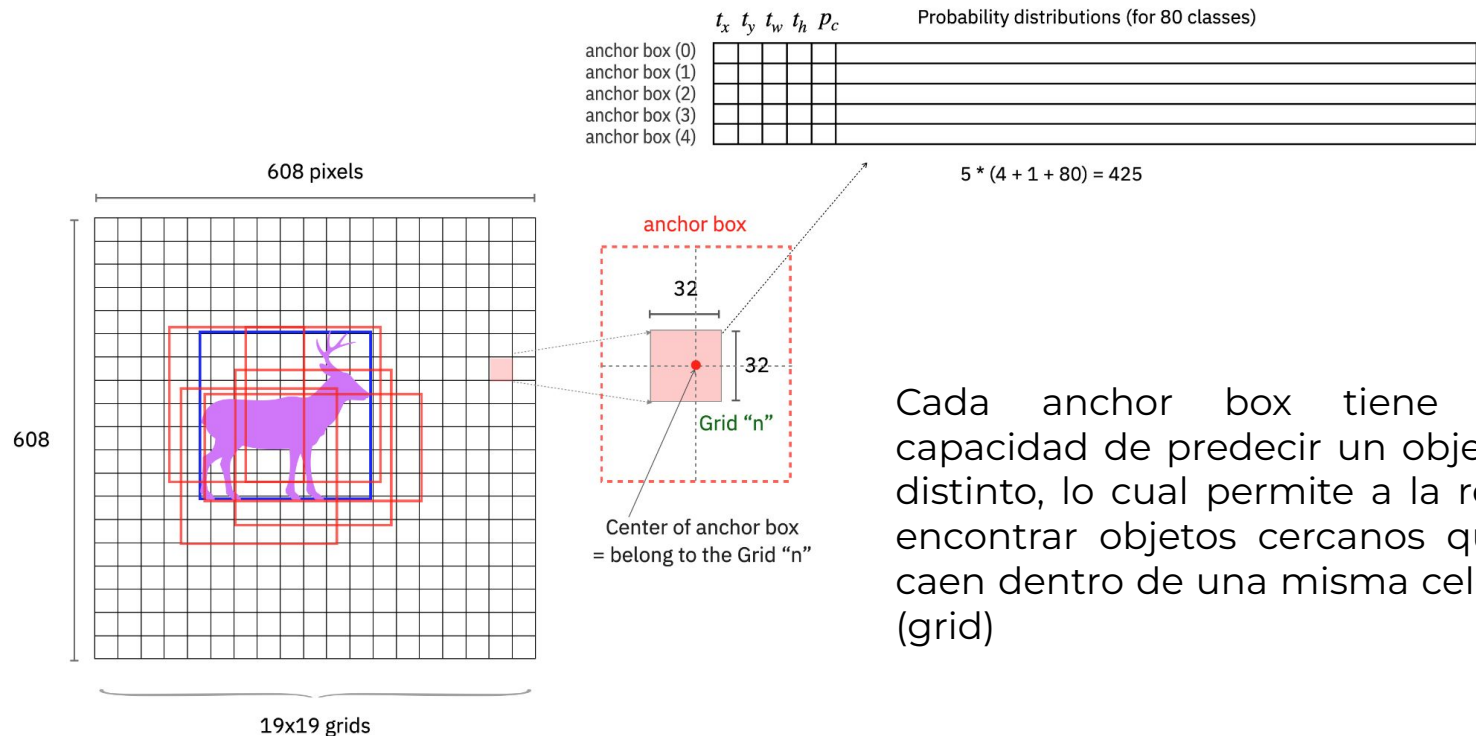
# Arquitecturas clásicas: Faster R-CNN

## Anchor Boxes

Son bounding boxes predefinidos que ayudan a la convergencia del entrenamiento de los modelos de detección de objetos. Por lo general, se predefinen distintos tamaños y relaciones de aspecto para los distintos anchors de cada una de las celdas del feature map de salida.

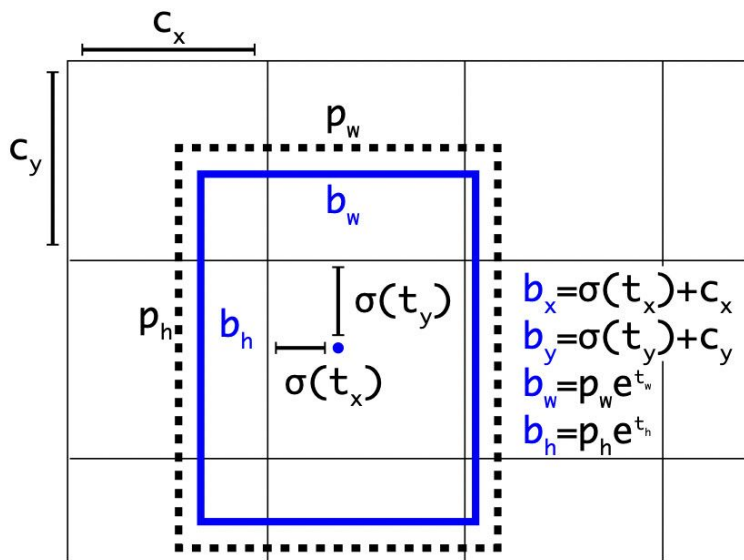


# Arquitecturas clásicas: Faster R-CNN



Cada anchor box tiene la capacidad de predecir un objeto distinto, lo cual permite a la red encontrar objetos cercanos que caen dentro de una misma celda (grid)

# Arquitecturas clásicas: Faster R-CNN



La red, en realidad, predice las coordenadas del centro del objeto en la celda correspondiente ( $t_x$  y  $t_y$ ), y el offset que existe entre el ancho y alto del ancho box y el ancho y alto real del objeto ( $t_w$  y  $t_h$ ).

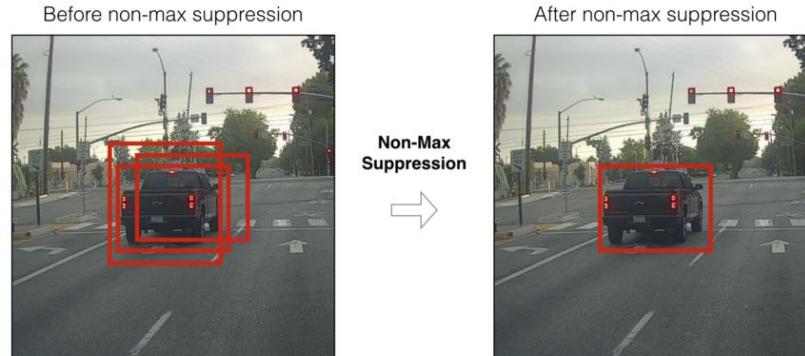
En la gráfica, la línea punteada corresponde al anchor box mientras que la línea azul es el bounding box generado por la red.

# Arquitecturas clásicas: Faster R-CNN

## Non-Max Suppression

Dado que los detectores de objetos son propensos a generar varios bounding boxes por cada objeto real de la imagen, resulta necesario utilizar algún método para **eliminar los boxes redundantes**. Este método es Non-Maximum Suppression (NMS).

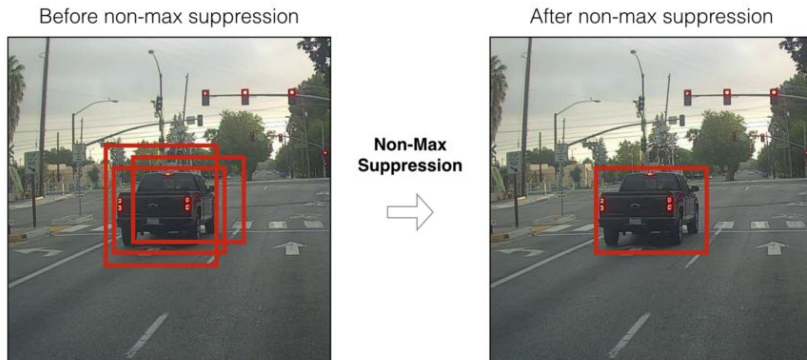
Para ello existen diferentes criterios, sin embargo, el más utilizado está basado en umbrales sobre la confianza en la detección y en el IoU entre bounding boxes predichas.



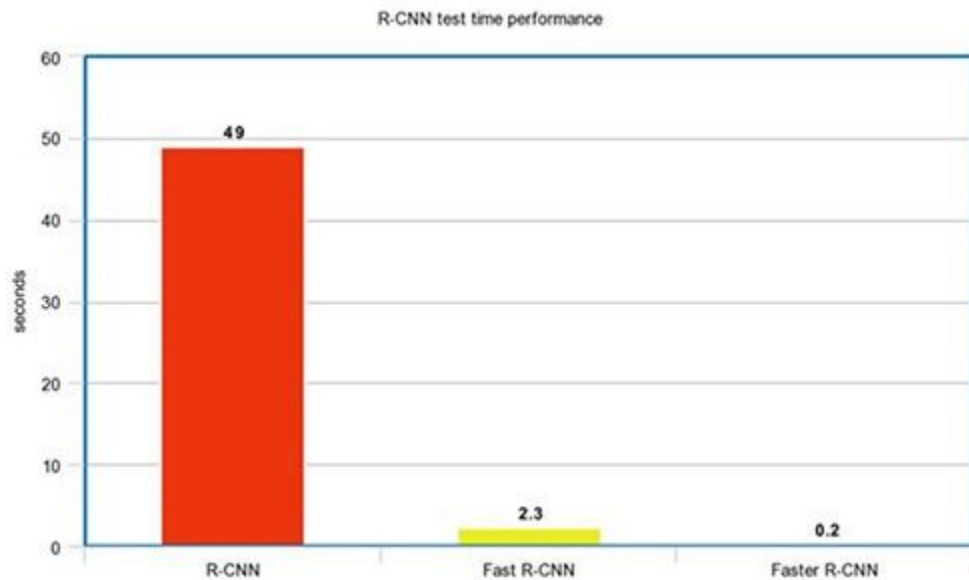
# Arquitecturas clásicas: Faster R-CNN

## Algoritmo

1. Ordenamos todas las detecciones de una clase según su valor de confianza.
2. Tomamos las dos detecciones con mayor confianza y computamos el IoU.
3. Si el valor de IoU es mayor a un umbral, descartamos la detección con menor confianza.
4. Repetimos el proceso desde el punto 1 hasta llegar al final de la lista.



# Comparativa



# Arquitecturas clásicas: YOLO

En 2016 se publicó el paper con la primer versión de YOLO, una red de detección de objetos capaz de realizar todo el procesamiento en un solo forward pass de la red. Es decir, a diferencia de los métodos de dos etapas, en este caso no existe un algoritmo o modelo que extraiga cuales son las regiones de interés sobre las cual buscar los objetos.

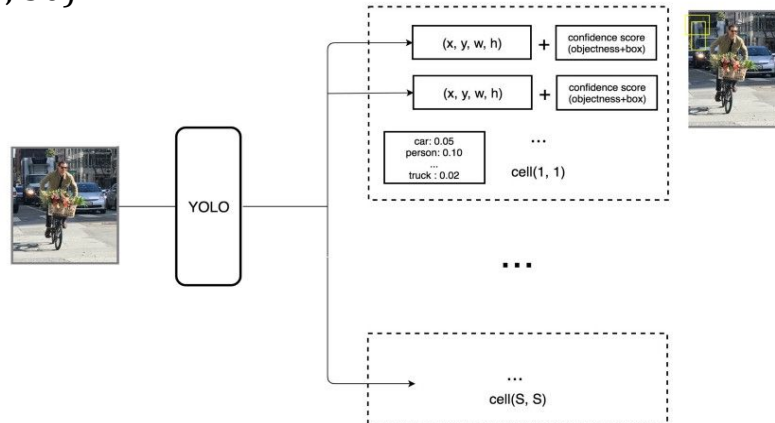
Por lo tanto, en YOLO (y en todos los detectores de una etapa) las predicciones de a qué clase pertenece el objeto y cuales son sus coordenadas de bounding box se realizan en la misma red convolucional.



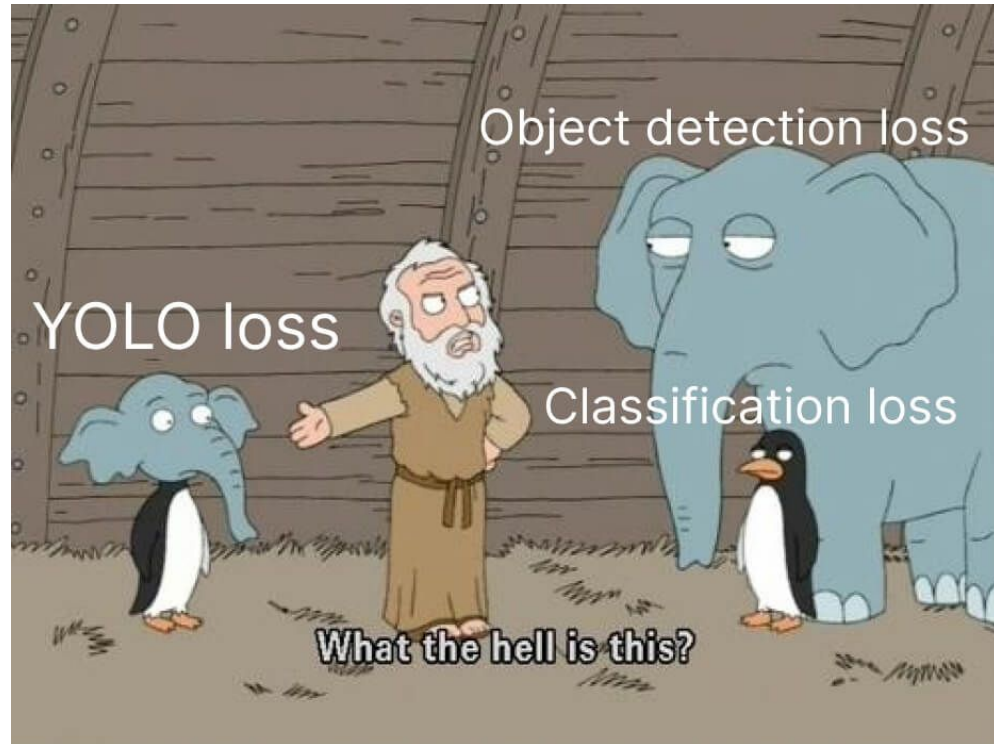


# Arquitecturas clásicas: YOLO

Para obtener las predicciones, la red genera un feature map de 7x7 celdas (PASCAL VOC). En cada celda predice 2 bounding boxes distintos, aunque solo es capaz de detectar un objeto en cada una. Cada predicción contiene 5 valores, las 4 coordenadas que lo definen y 1 valor que refleja que tan seguro está de que exista un objeto dentro de dicho box. Además, devuelve las probabilidades de cada clase, 20 valores en este caso. Por lo tanto, una predicción de YOLO tiene la forma:  
 $(7, 7, 2 \times 5 + 20) = (7, 7, 30)$



# Arquitecturas clásicas: YOLO



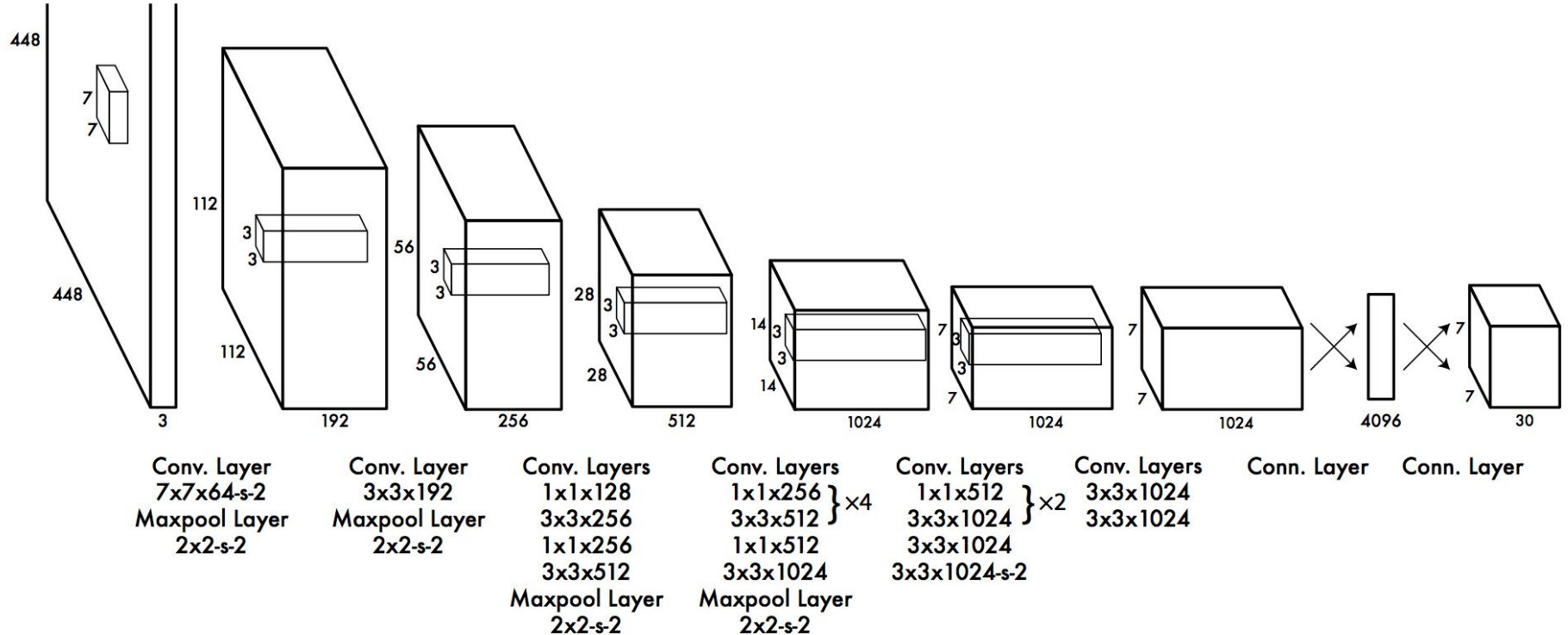
# Arquitecturas clásicas: YOLO

Yolo utiliza la suma de los cuadrados de los errores entre las predicciones y los ground truth para calcular el error. Dado que YOLO predice varias bounding boxes, para determinar cuál es el true positive se computa el IoU de los boxes predichos con el ground truth y se selecciona el box que da mayor valor. La función de error tiene la siguiente forma:

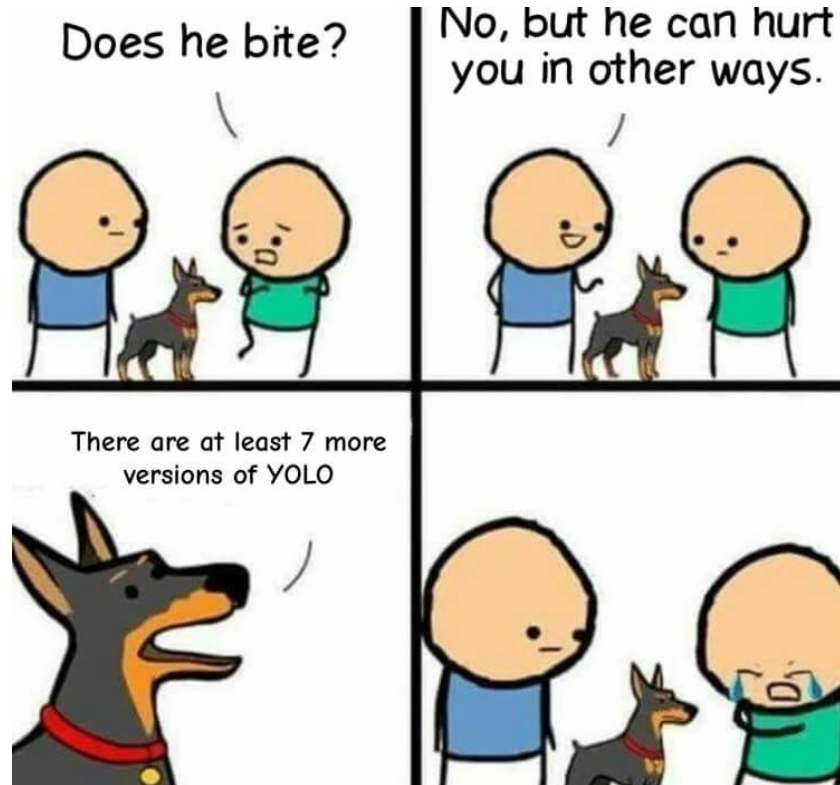
El primer y segundo término hacen referencia al error de localización, el cual solo se computará para el BB asignado al ground truth. El tercer y cuarto término hacen referencia al error de confianza con que un objeto se encuentra dentro del BB. Por último, el quinto término se refiere al error de clasificación del objeto detectado.

$$\begin{aligned} & \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[ (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\ & + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[ \left( \sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left( \sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\ & + \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} (C_i - \hat{C}_i)^2 \\ & + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{noobj}} (C_i - \hat{C}_i)^2 \\ & + \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \end{aligned}$$

# Arquitecturas clásicas: YOLO



# Arquitecturas clásicas: YOLO



# Arquitecturas clásicas: YOLOs



## **YOLOv2/YOLO9000 - 2016**

Incorpora anchor boxes y aumenta la resolución. También permite detectar más de un objeto por celda. Mejora mucho en precisión de detección, sobre todo en los grupos de objetos más pequeños.

## **YOLOv3 - 2018**

Incorpora conexiones residuales y más capas en el extractor de features. Para realizar las predicciones toma features de 3 capas distintas del backbone y aumenta los anchor boxes a 9. Mejora la precisión en objetos pequeños.

## **YOLOv4 - 2020**

Se introducen varios cambios en el proceso de entrenamiento, augmentacion de datos, regularizaciones, etc. Es la primera versión de la red que cambia de autor.

## **YOLOv5 - 2020**

Implementada totalmente en Pytorch, es la única implementación que no tiene paper aunque su repositorio está muy bien documentado. Fue desarrollada por una empresa llamada Ultralytics. Presenta cambios similares a los implementados en YOLOv4 y cuenta con 4 tamaños: small, medium, large y extra large.

# Arquitecturas clásicas: YOLOs



## **YOLOv6 -2022**

Desarrollada por un grupo independiente, está orientada a aplicaciones industriales y edge computing. Incorpora técnicas como asignación de etiquetas, [self-distillation](#), más épocas de entrenamiento, quantization. Además usa una arquitectura [RepVGG](#) como backbone.

## **YOLOv7 - 2022**

Nuevo paper desarrollado por otro grupo que logra uno de los mejores mAP sobre COCO hasta el momento. Incorpora muchas mejoras en la arquitectura, función de error, entrenamiento, etc.

## **YOLOv8 - 2023**

Nueva versión desarrollada por Ultralytics sin paper pero con buena documentación en su [web](#).

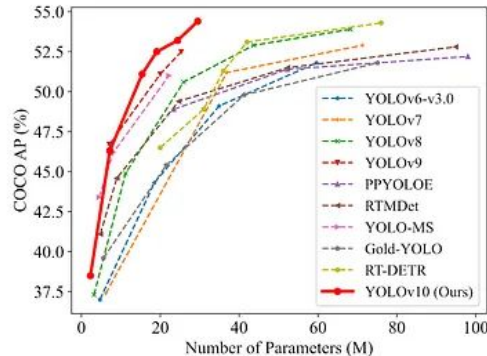
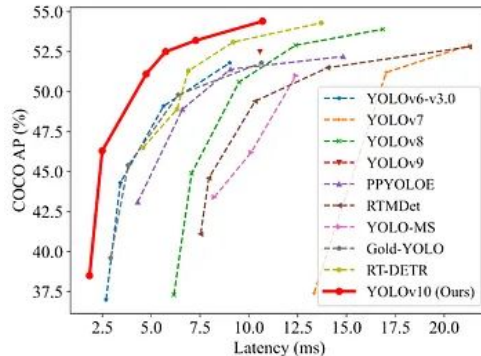
# Arquitecturas clásicas: YOLOs

## YOLOv9 -2024

Desarrollada por un grupo de investigadores de Taiwan usando la base de código de Ultralytics. Introduce un enfoque innovador para atacar los problemas más comunes de las redes de detección de objetos. Los puntos más destacables son el uso de funciones reversibles, PGI y GELAN.

## YOLOv10 -2024

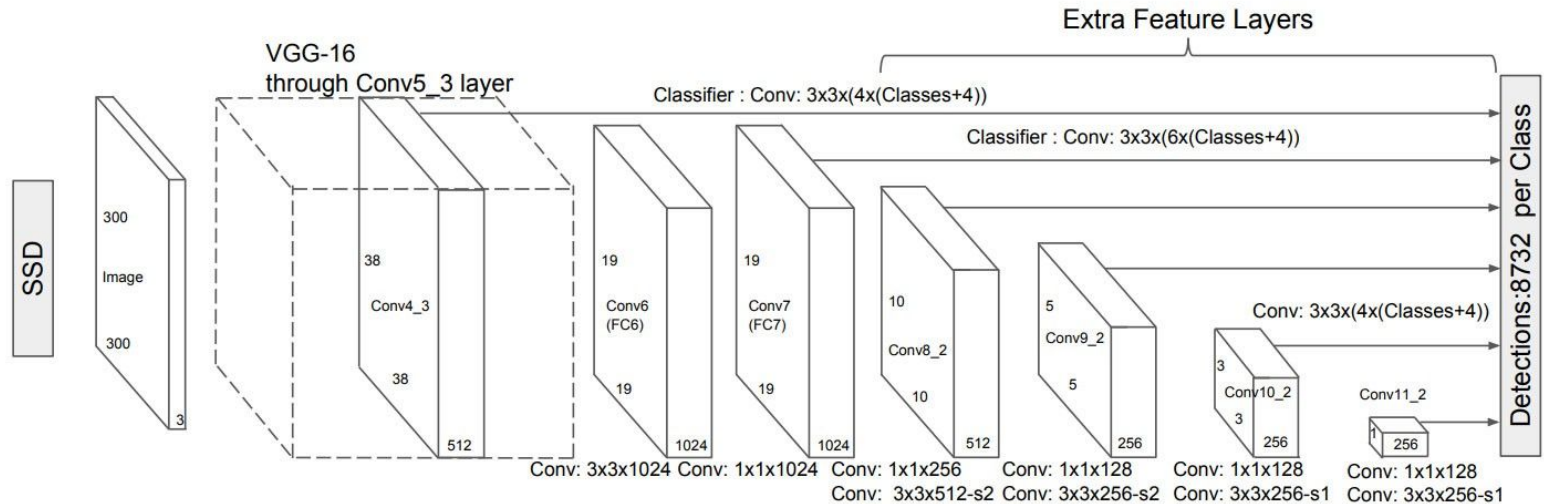
Desarrollada por un grupo de China. Entre sus aspectos más destacables está la eliminación de NMS.





# Arquitecturas clásicas: SSD

En el año 2016 también se publicó el paper que presentaba SSD (Single Shot Detector). Esta arquitectura de una etapa presentaba mejoras respecto a YOLO en cuanto a tiempo de inferencia y mAP obtenido.



# Arquitecturas clásicas: SSD



La estructura de esta red está compuesta por dos partes: un **extractor de features**, implementado con una VGG16, y **capas convolucionales** para la detección de objetos. Es decir, que la regresión necesaria para calcular las coordenadas de los bounding boxes es realizada por las mismas convoluciones.

Por ejemplo, luego de la capa Conv4\_3, la salida es de 38x38x512. A este volumen se le aplican filtros de 3x3 para realizar 4 predicciones por cada celda de dicho volumen. Una predicción, está compuesta por los 4 números que representan el bounding box, más las  $N+1$  clases del dataset en cuestión. De ahí que la cantidad de filtros queda:  $4 \times ((N+1) + 4)$ .

Si se utiliza el dataset PASCAL VOC, solo en esta capa, la red está realizando 5776 predicciones.

# Arquitecturas clásicas: SSD

## Multi-Scale Detection

Para mejorar la precisión en las detecciones de distintos tamaños de objetos, esta red cuenta con 6 capas convolucionales para realizar las predicciones. Estas capas van reduciendo la dimensionalidad espacial de forma tal que los últimos features maps cuentan con menor resolución, lo que les permite detectar objetos más grandes dentro de la imagen. Por el contrario, las detecciones generadas por las primeras capas están más enfocadas en los objetos más pequeños.

Prediction source layers from:						mAP		# Boxes
conv4_3	conv7	conv8_2	conv9_2	conv10_2	conv11_2	use boundary boxes?		
Yes	No							
✓	✓	✓	✓	✓	✓	74.3	63.4	8732
✓	✓	✓	✓	✓		<b>74.6</b>	63.1	8764
✓	✓	✓	✓			73.8	68.4	8942
✓	✓	✓				70.7	69.2	9864
✓	✓					64.2	64.4	9025
	✓					62.4	64.0	8664

Table 3: Effects of using multiple output layers.

# Arquitecturas clásicas: SSD



- Para facilitar la predicción de los bounding boxes, se utilizan **anchor boxes**. Se entrenó a la red de forma tal que prediga el offset con respecto al anchor box correspondiente, en cada predicción.
- Utiliza una **función de error que toma en consideración tanto el error de localización como el de clasificación**. En general, se toman como ejemplos positivos aquellos en los que el anchor box tiene un IoU mayor a 0.5 con el bounding box etiquetado.
- Debido a la gran cantidad de predicciones que realiza la red, existen muchas más predicciones erróneas que acertadas. Esto genera un **desbalance de clases en el entrenamiento**. Para evitar este problema, durante el entrenamiento se toman las predicciones negativas que tienen el mayor error, manteniendo siempre una relación de 3:1 entre predicciones negativas y positivas.

# Arquitecturas clásicas: RetinaNet



Otra arquitectura de una etapa es RetinaNet. En ella se proponen dos grandes mejoras respecto a las demás arquitecturas de una etapa existente hasta este momento:

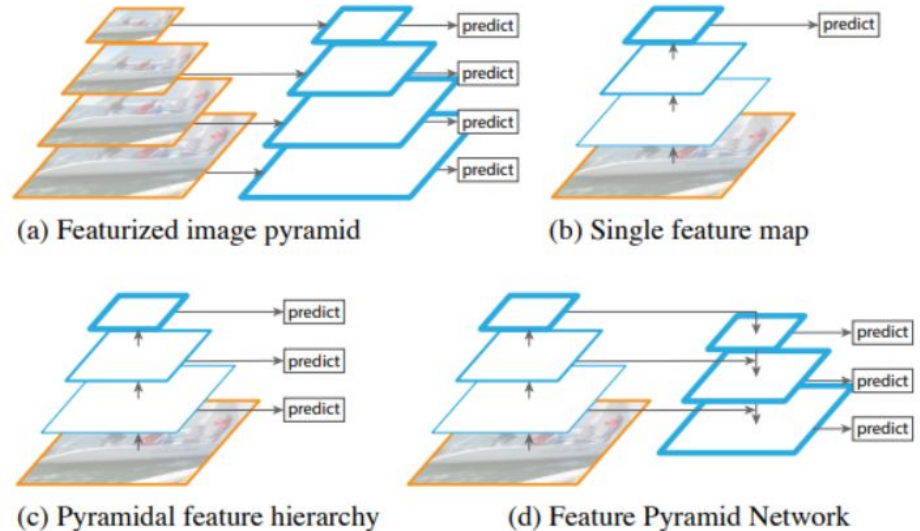
- **Feature Pyramid Network (FPN):** Colocaron una arquitectura similar a la propuesta en el paper original, sobre una ResNet-50 para realizar predicciones a partir de features maps de múltiples escalas. Este mejora drásticamente las detecciones, sobre todo en objetos pequeños.
- **Focal Loss:** Modificaron la parte de la función de error correspondiente a la clasificación para mejorar la influencia de las predicciones menos seguras, sobre el valor final del error.

# Arquitecturas clásicas: RetinaNet

## Feature Pyramid Network (FPN)

La arquitectura tiene un camino desde abajo hacia arriba (ResNet-50) donde se extraen features y se reduce la resolución. Luego, hay un camino desde arriba hacia abajo, con conexiones laterales, donde se hace un upsampling de los features maps y se los mezcla con los features maps anteriores.

Luego, para la predicción de clases y bounding boxes, existen dos sub redes convolucionales que se encargan de la clasificación y regresión en cada nivel.

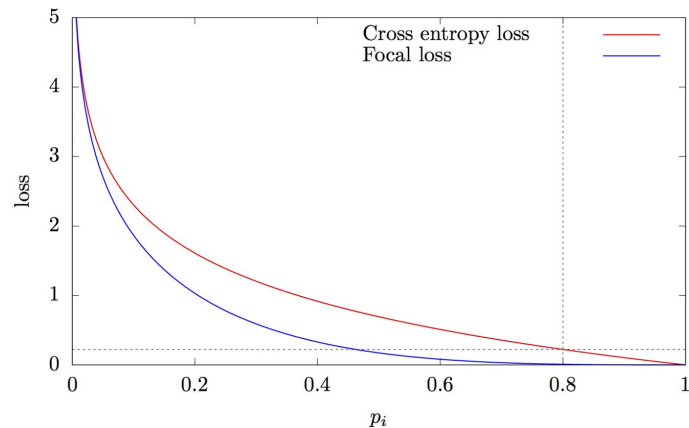


# Arquitecturas clásicas: RetinaNet

## Focal Loss

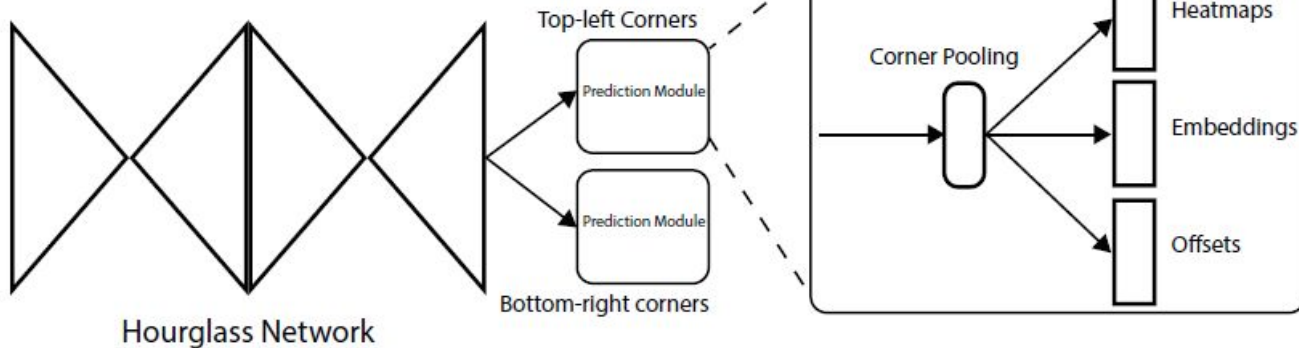
Es una variante de la función de cross entropy loss para restarle importancia a la gran cantidad de predicciones negativas que puede hacer la red, que tienen un porcentaje de confianza bastante alto. De esta forma, la suma de los pequeños aportes de todas esas predicciones no sobrepasa el error generado por las pocas predicciones positivas que tienen porcentajes de confianza más bajos.

$$C(p, y) = - \sum_i y_i (1 - p_i)^\gamma \ln p_i$$



# CornerNet

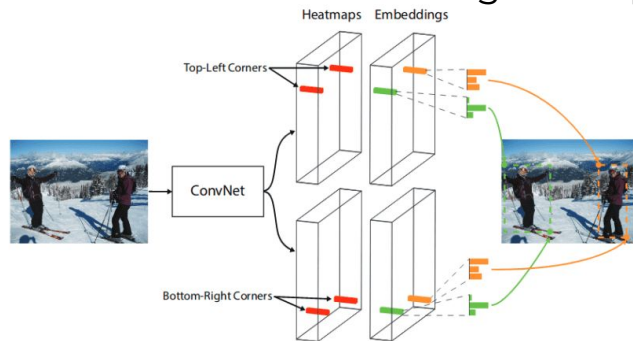
La arquitectura de esta red busca detectar a los bounding boxes de los objetos como un par de puntos, los vértices superior izquierdo y el inferior derecho, **eliminando la necesidad de utilizar anchor boxes!**





# CornerNet

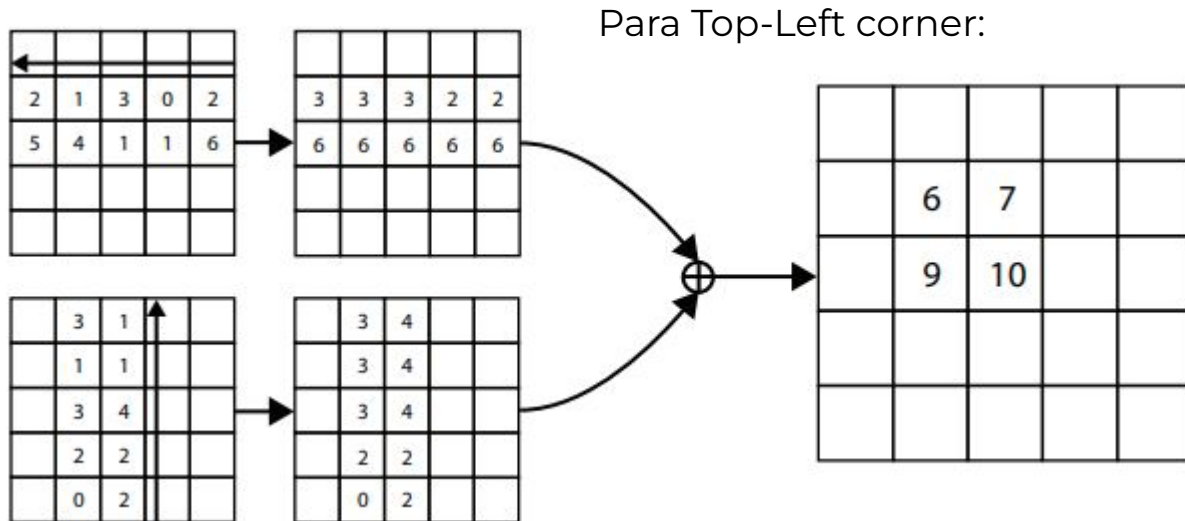
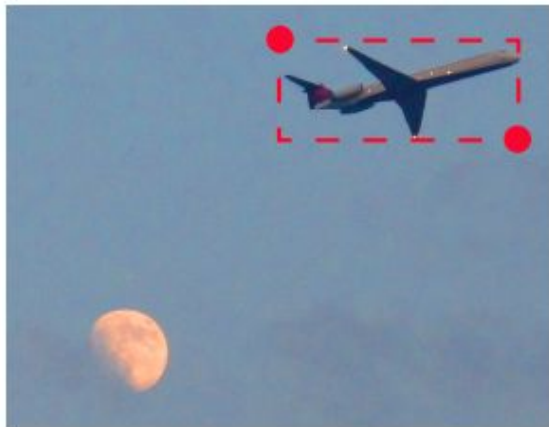
- Como red convolucional para extraer features se utiliza una arquitectura tipo hourglass.
- Hay dos módulos de predicciones, uno para las esquinas superiores izquierdas de los objetos y otro para las esquinas inferiores derechas.
- Cada módulo contiene una capa de **corner pooling** que alimenta las predicciones de heatmaps, embeddings y offsets.
- Los heatmaps indican las clases de las predicciones. La salida tiene  $c$  canales, donde  $c$  es la cantidad de clases. No hay clase de background.
- Los embeddings están entrenados para poder asociar los corners de un mismo objeto, devuelto por cada predictor.
- Los offsets sirven para ajustar finamente los bounding boxes predichos.



# CornerNet

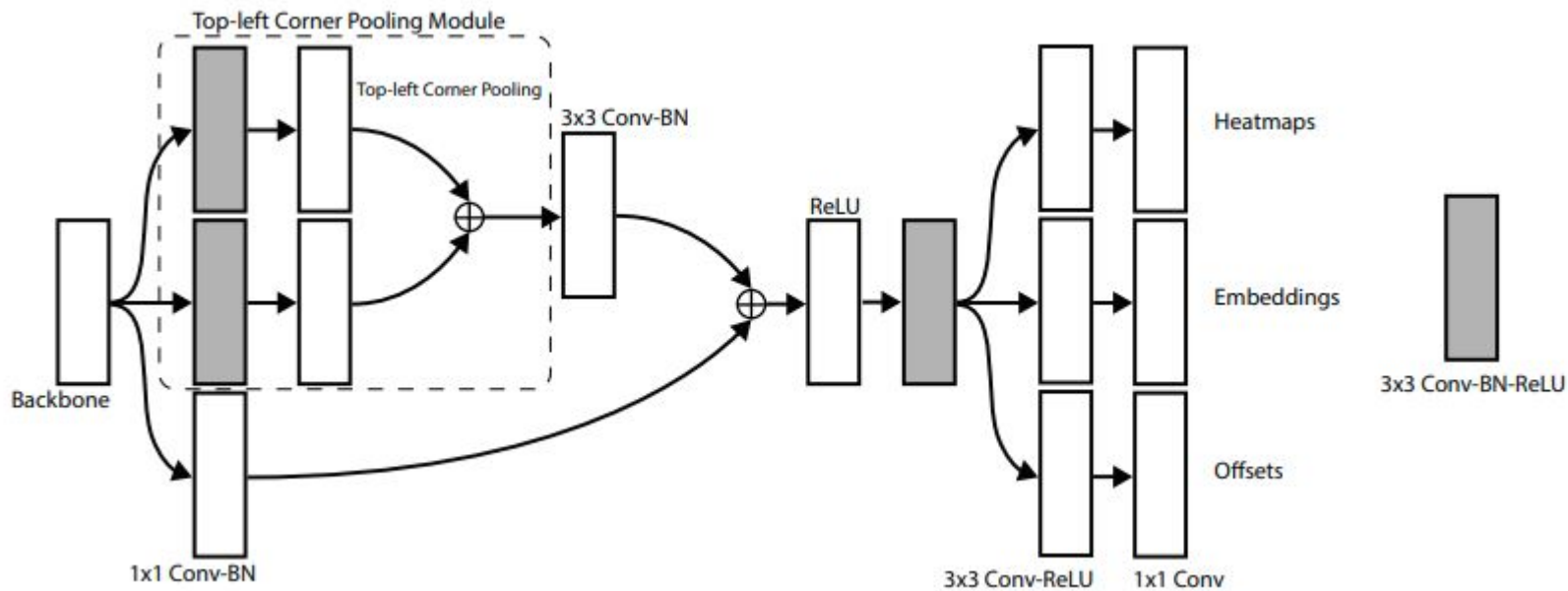
## Corner Pooling

El problema de este tipo de redes es que las corners de los bounding boxes no suelen estar en lugares de la imagen donde haya contexto sobre el objeto en cuestión. Para solventar este problema, se introduce la capa de corner pooling.



# CornerNet

Arquitectura de un bloque de predicciones:



# Arquitecturas clásicas: Otras



Además de las vistas en la clase, existen otras arquitecturas que buscan resolver el problema de la detección de objetos. Algunas de estas son:

- FPN: [Link](#)
- R-FCN: [Link](#)
- YOLO v3: [Link](#)
- YOLO v4: [Link](#)
- YOLO v5: [Link](#)
- YOLO v6: [Link](#)
- YOLO v7: [Link](#)
- YOLO v8: [Link](#)
- YOLOX: [Link](#)
- EfficientDet: [Link](#)
- CenterNet: [Link](#)

# Desbalance de datos



Cuando se trabaja con un problema de detección de objetos pueden existir distintos tipos de desbalance de datos dentro del dataset:

- Entre clases y fondo
- Entre cantidad de instancias de cada clase
- Entre la ubicación de cada clase en la imagen
- Entre el tamaño de los objetos de cada clase
- Entre relación de aspecto de los objetos de cada clase

# Etiquetado de Bounding Boxes

Es importante tener en cuenta algunos puntos a la hora de hacer etiquetado de bounding boxes en imágenes para no cometer errores que perjudiquen la performance de la red.

## ***Ajustar al máximo los bounding boxes***

Es recomendable que los bounding boxes se ajusten al máximo con el tamaño del objeto, es decir, que no queden píxeles correspondientes al fondo entre el BB y el objeto de interés.



# Etiquetado de Bounding Boxes



## ***Etiquetar todos los objetos de interés***

Siempre se deben etiquetar todos los objetos de interés que se encuentren dentro de la imagen, incluso si estos no se encuentran en condiciones “normales”. De no ser así, la red va a utilizar esa parte de la imagen para entrenar la categoría de background, lo cual termina confundiendo al modelo.

## ***Objetos parcialmente ocultos***

En estos casos se recomienda etiquetar a los objetos como si estuvieran completamente visibles, en lugar de solo la porción que se ve. Incluso en los casos que exista un poco de superposición con otro objeto a etiquetar.

# Etiquetado de Bounding Boxes

## ***Objetos en diagonal***

Tienen la desventaja de que ocupan muy poco espacio en relación con el BB necesario para etiquetarlos por lo que lo que predomina es el background. Esto puede hacer que la red interprete dicho BB como background en lugar de objeto. Esto se compensa con suficiente cantidad de datos.





# Etiquetado de Bounding Boxes



## ***Tiempo de etiquetado promedio***

Se estima que para una persona inexperta, el tiempo de etiquetado promedio de **una** imagen puede ser entre 1 a 3 minutos, dependiendo de la cantidad de objetos en la imagen. Este tiempo tiende a disminuir con la práctica.

## ***Herramientas de etiquetado***

- [Roboflow](#)
- [V7](#)
- [SuperAnnotate](#)
- [CVAT](#)
- [labelImg](#)

y muchas otras más...

# Formatos de etiquetas



Lamentablemente no existe un único formato estándar adoptado por la comunidad para representar las etiquetas de un problema de detección de objetos. Sin embargo, los 3 principales son: COCO format, Pascal VOC format y YOLO format.

<https://roboflow.com/formats>

La misma plataforma de Roboflow nos permite transformar las etiquetas de un formato a otro.

Por otro lado, si queremos transformar etiquetas entre distintos tipos de problemas, por ejemplo, etiquetas hechas para segmentación de instancias a etiquetas para detección de objetos, podemos usar la librería Supervision de Roboflow.

<https://supervision.roboflow.com/latest/>

# Ejemplos prácticos



Para realizar entrenamientos sobre distintas versiones de redes de detección de objetos podemos utilizar la plataforma y el código que nos provee <https://roboflow.com/>

Otras opciones pueden ser:

- [Tensorflow Object Detection API](#)
- [Detectron2](#)

No es obligatorio pero si quieren dejar su feedback sobre esta clase pueden hacerlo aquí:

[Encuesta sobre la clase](#)