Trabajo de Laboratorio N°5:

YOLO

Materia: Inteligencia Artificial.

Carrera: Ingeniería en Informática.

Docente: Ing. Federico Gabriel D´Angiolo.

Alumno: Lew Imanol

Índice

1-Objetivo

El objetivo de este trabajo consiste en analizar y detallar el algoritmo YOLO (you only look once), “solo miras una vez” en español, el cual tiene como objetivo detectar objetos en imágenes y videos. No solo detectara los objetos sino que también marcará donde estos se encuentran en las imágenes (**frames**) e incluso los podría clasificar. Su nombre hace referencia a su funcionamiento interno, en el que las imágenes “solo se ven una única vez” en todo el algoritmo.

2-introducción

2.1- Aplicaciones

El punto fuerte de YOLO es su magnífica velocidad, pero para archivar esta velocidad debe sacrificar precisión. No obstante, su precisión es lo suficientemente buena como para ser utilizado en aplicaciones reales. A partir de la versión YOLOV3 la precisión se incrementa en gran medida, rivalizando con las de los algoritmos con los cuales se lo suele comparar (familia de algoritmos R-CNN).

YOLO tiene lugar en aplicaciones de detección de objetos en tiempo real, como por ejemplo en pilotos automaticos de automoviles, deteccion de anomalias especificas (ej:que una persona no lleve barbijo en cuarentena).

2.2 - Conceptos

Previo a hablar de lleno sobre el algoritmo, se necesitaran explicar ciertos conceptos.

Ground truth y Bounding box

Estos 2 conceptos se los pueden imaginar como la información etiquetada o **label** (*Ground truth*) y la información predicha (*Bounding box*). Se presenta un ejemplo de esto en la imagen 1.

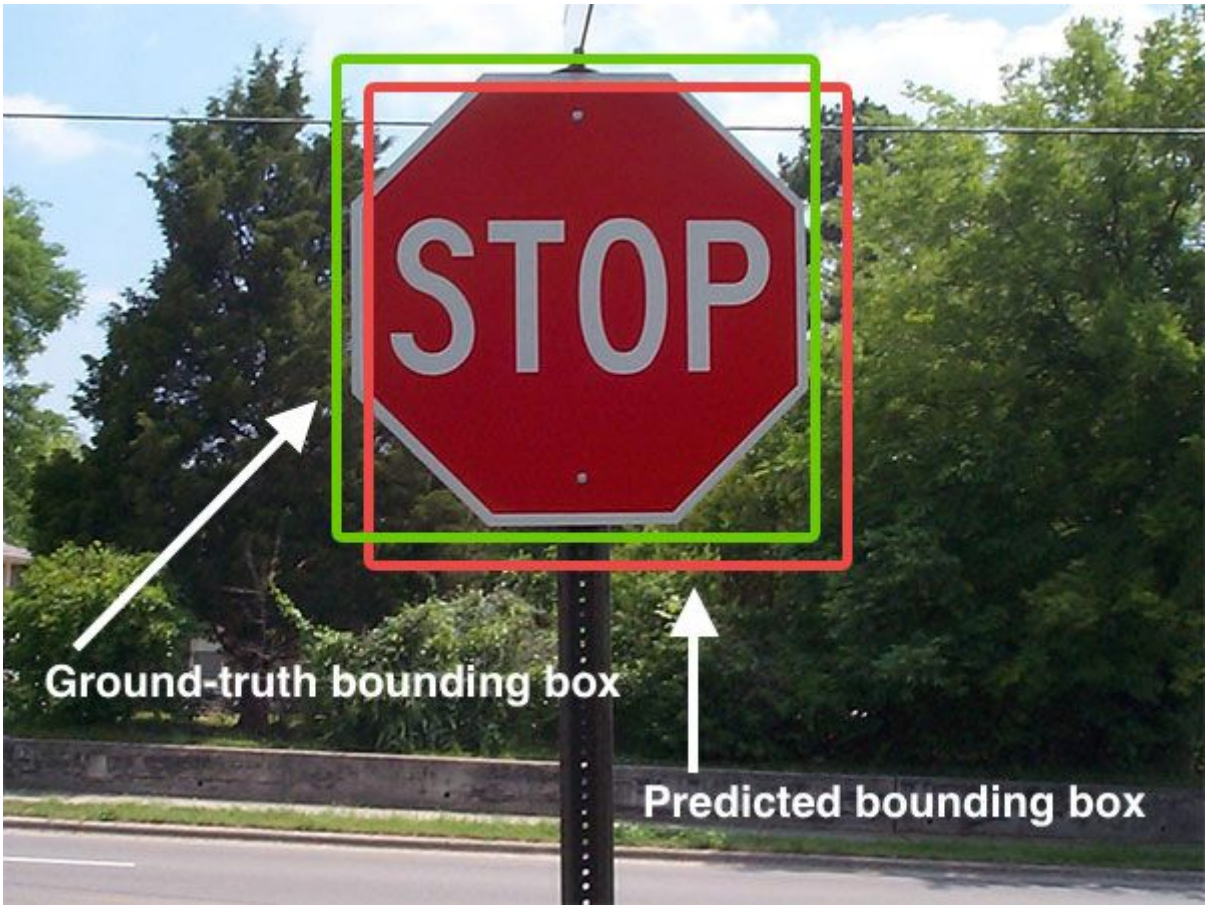

imagen 1 - ejemplo de ground-truth y bounding box

Para el caso de la imagen 1 el *Ground truth* (recuadro verde) sería la señal de stop encuadrada perfectamente, donde dicho label fue previamente creado manualmente por un usuario. En cambio, el *Bounding box* (recuadro rojo), que es una predicción es lo que el algoritmo detecta como objeto. Los algoritmos pueden (y lo harán) dar varias bounding box para un mismo objeto. El como solucionarlo se verá más adelante por medio de la estrategia *non-maxima suppression*.

Intersection over union

A continuación, un concepto que mezcla los anteriores (ground truth y bounding box). La *intersección sobre union* o **IoU** es una métrica que relaciona los 2 conceptos previos. Esta métrica mide que tan bueno es el resultado de la bounding box

Al igual que en la matematica, hablamos de union e interseccion, pero llevado al campo de las imágenes.

Intersección

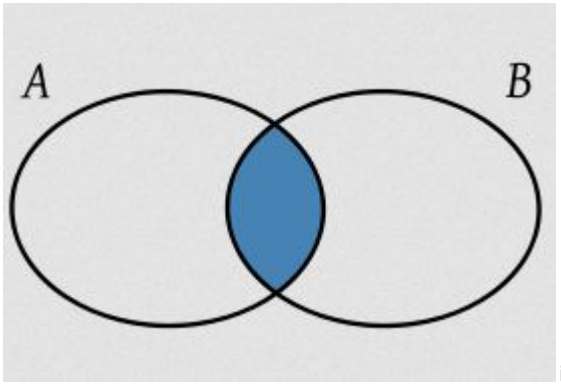

imagen 2 - intersección

Unión

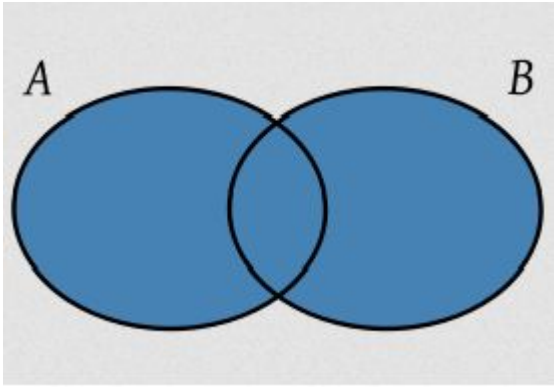

imagen 3 - Unión

La IoU es en sí una relación entre estos 2 conceptos

$$IoU = \frac{intersección}{unión}$$

Si lo llevaramos a nuestro ejemplo de la señal de stop quedaría de la siguiente forma:

imagen 4 - ejemplo cálculo de IoU

donde:

Area of overlap es la intersección entre el ground truth y el bounding box

Area of union es la unión entre ground truth y bounding box

Realizando la relación entre estas 2 áreas generamos la metrica IoU. En el mejor de los casos, tendríamos al bounding box encima del ground truth, por lo que el área de intersección sería igual al área de estos cuadrados y por lo tanto el área de la unión de estos sería exactamente igual a la intersección dando como resultado 1. En el peor de los casos, tendríamos las 2 cajas sin ninguna intersección, haciendo que este valor sea 0, y el área de la unión sería la suma de las 2 áreas. El valor final en este caso sería 0.

Non-maxima suppression

Non-maxima suppression o **NMS** es una técnica para filtrar bounding boxes. Se pueden utilizar varios criterios para este filtrado, como ***confidence score*** (que tan seguro es que haya un objeto en la bounding box predicha) y ***overlap threshold*** (que tanto se pegan las bounding boxes).

El NMS se realiza de la siguiente forma. Tiene como entrada una lista de propuestas, donde las propuestas en sí son posibles objetos candidatos a ser detectados, que vienen acompañados de un *confidence score* y un *overlap threshold* . La salida es otra lista de objetos candidatos pero filtrados.

Funcionamiento del filtro: Se obtiene de la lista el candidato A con más *confidence score* y se lo agrega a la lista de salida. Se compara su IoU con el resto de los candidatos B uno por uno. Si el IoU del candidato A con el candidato Bi supera un *overlap threshold*, este candidato Bi se presume que detectó el mismo objeto que el candidato A, por lo que es removido de los candidatos a detectar, sacándolo de la lista de entrada. Si el IoU no supera el *overlap threshold* se presume que este candidato Bi está detectando un objeto distinto y se lo agrega a la lista de salida. El proceso se vuelve a repetir para cada candidato en la lista de salida hasta que no haya candidatos en la lista de entrada.

Imagen con el antes y después de aplicar NMS

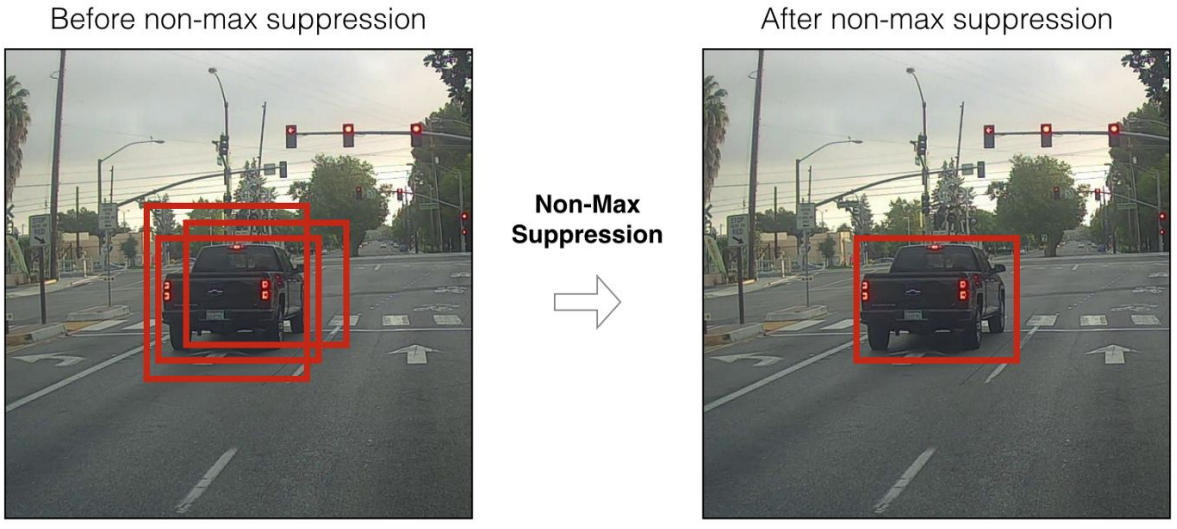

imagen 5 - ejemplo de NMS

Precisión mAP

La precisión mAP es una métrica de evaluación, utilizada en los algoritmos de reconocimiento de objetos. Sus siglas representan *"mean average precision".* Es un valor que va de 0 a 100 y por supuesto que a mayor valor, mejor la evaluación. Para obtener el valor, lo que se hace es calcular previamente otro valor, llamado *"average precision"* (**AP**) para cada clase distinta perteneciente al dataset de imágenes con el que se realiza la evaluación. Este valor nos indicará la precisión para la detección de cada clase. A su vez, mAP es el promedio de **AP** para todas las clases.

3 - Redes Neuronales Convolucionales (CNN)

Como YOLO usa CNNs internamente, se explicará el funcionamiento de estas de manera resumida.

La CNN es un tipo de Red Neuronal Artificial de aprendizaje supervisado que procesa sus capas imitando al córtex visual del ojo humano para identificar distintas características en las entradas que en definitiva hacen que pueda identificar objetos y "ver". Para ello, la CNN contiene varias capas ocultas especializadas y con una jerarquía: esto quiere decir que las primeras capas pueden detectar líneas, curvas y se van especializando hasta llegar a capas más profundas que reconocen formas complejas como un rostro o la silueta de un animal.

La CNN en la capa de entrada tomara los pixeles de una imagen. Si la imagen es de 28x28, entonces se necesitarán 28x28=784 neuronas. Aquí se asume que la imagen está en una escala de grises (el valor de cada pixel va de 0 a 255). Si no fuera el caso y estuviera a color, se utilizarían 3 canales (RGB) por lo que las neuronas de entrada sumarían una cantidad de 28*28*3=2352.

Previo a lanzar los valores de los pixeles a la CNN estos se deben normalizar, por lo que sus valores son divididos entre 255, haciendo que nuestros valores de entrada vayan de 0 a 1.

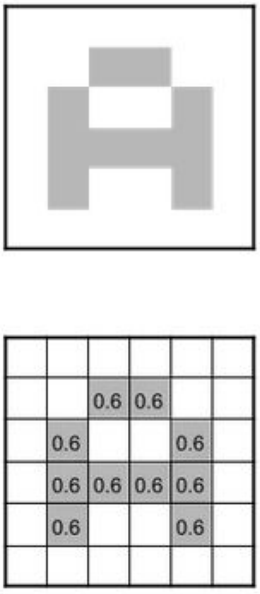

imagen 6 - explicación CNN

Una vez normalizados los valores, se realiza la etapa de convoluciones, lo que consiste en tomar pequeños grupos de píxeles de la imagen de entrada e ir operando matemáticamente (producto escalar) contra varias pequeñas matrices llamadas kernel. Esas matrices kernel supongamos a modo de ejemplo que son de tamaño 3×3 pixels y recorren todas las neuronas de entrada (de izquierda-derecha, de arriba-abajo) y generan una nueva matriz de salida, que en definitiva serán nuestra nueva capa de neuronas ocultas.

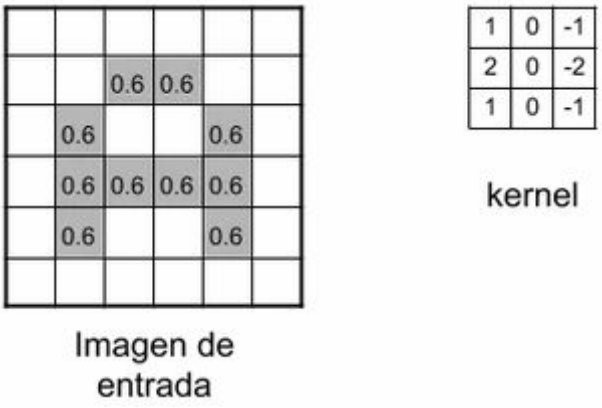

imagen 7 - explicación cnn

A modo de ejemplo, se tendrán 32 kernels (el conjunto de kernels tiene como nombre **filtro**), y por lo tanto, tendremos como resultado 32 matrices de salida (a estas matrices de salida se les llama *feature mapping*) lo que da como resultado tener una primer capa oculta de neuronas con una cantidad equivalente a 28*28*32=25088. En esta primer convolución, es como si obtuviéramos 32 imágenes filtradas nuevas. Estas imágenes nuevas lo que están “dibujando” son ciertas características de la imagen original. Esto ayudará en el futuro a poder distinguir un objeto de otro (por ej gato o perro).

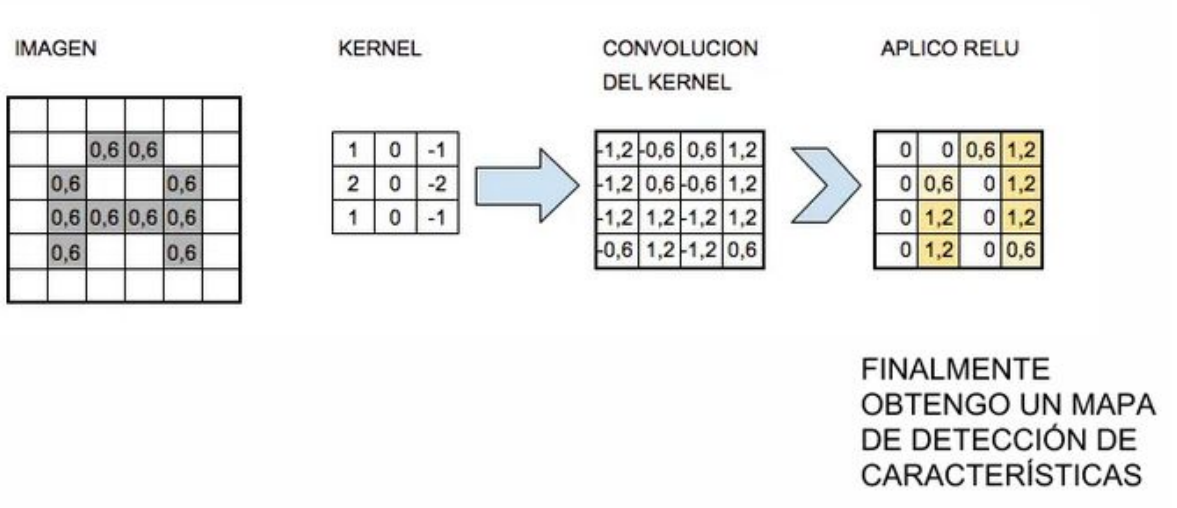

imagen 8 - explicación CNN

A continuación, se procede con la etapa de *Subsampling,* que básicamente consiste en reducir la cantidad de neuronas antes de realizar otra convolución. Esto se hace porque es muy sencillo que la cantidad de neuronas crezca de manera abrupta.

El *subsampling* consiste en reducir el tamaño de nuestras imágenes filtradas y a su vez, que prevalezcan las características más importantes que detecto cada filtro. Hay varios métodos de *subsampling*, en este ejemplo se utilizara el método *max-pooling*.

El método max-pooling consiste de quedarse con las características más importantes a partir de regiones, tal como muestra la imagen 9 (básicamente, la característica más importante es el valor más alto en la subregión)

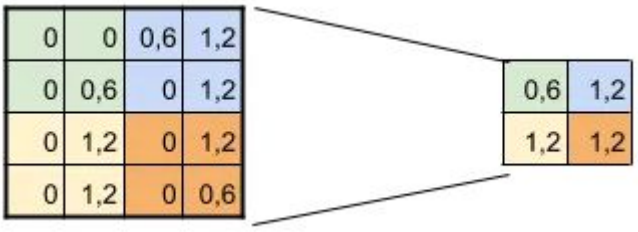

imagen 9 - explicación max-pooling

Tal como en la imagen, en nuestro caso se seguirá con un max-pooling de 2x2. Esto quiere decir que recorreremos cada una de nuestras 32 imágenes de características obtenidas anteriormente de 28x28 de izquierda-derecha, arriba-abajo e iremos preservando el valor más alto de entre esos 4 pixeles. Usando 2×2, la imagen resultante es reducida a la mitad y quedará de 14×14 pixeles. Luego de este proceso de subsampling nos quedarán 32 imágenes de 14×14, pasando de haber tenido 25.088 neuronas a 6272. Son bastantes menos y en teoría siguen almacenando la información más importante para detectar características deseadas.

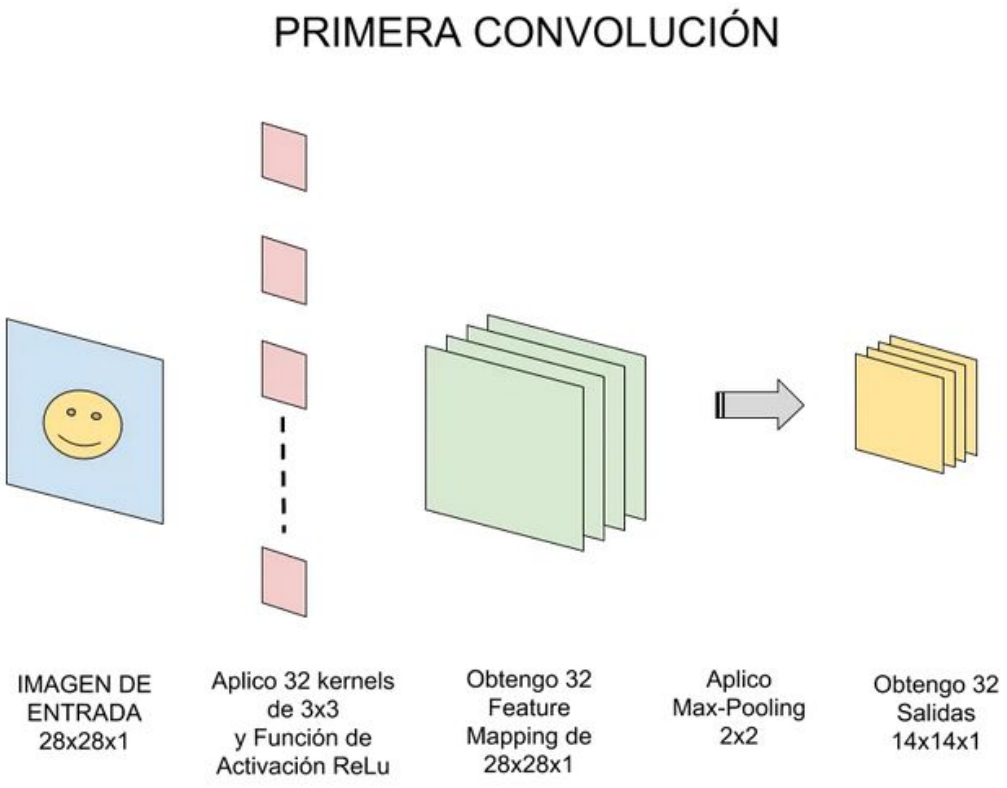

imagen 10 - ilustración primera convolución

La primer convolución es capaz de detectar características primitivas como líneas o curvas. A medida que se hagan más capas con las convoluciones, los mapas de características serán capaces de reconocer formas más complejas.

Segunda convolución (y sucesivas):

En la entrada de la segunda convolución se tendrá la salida de la primer convolución, 14x14x32. Luego se le aplican 64 kernels, se obtiene el feature mapping, se aplica nuevamente max-pooling de 2x2 y se tiene a la salida 7x7x64

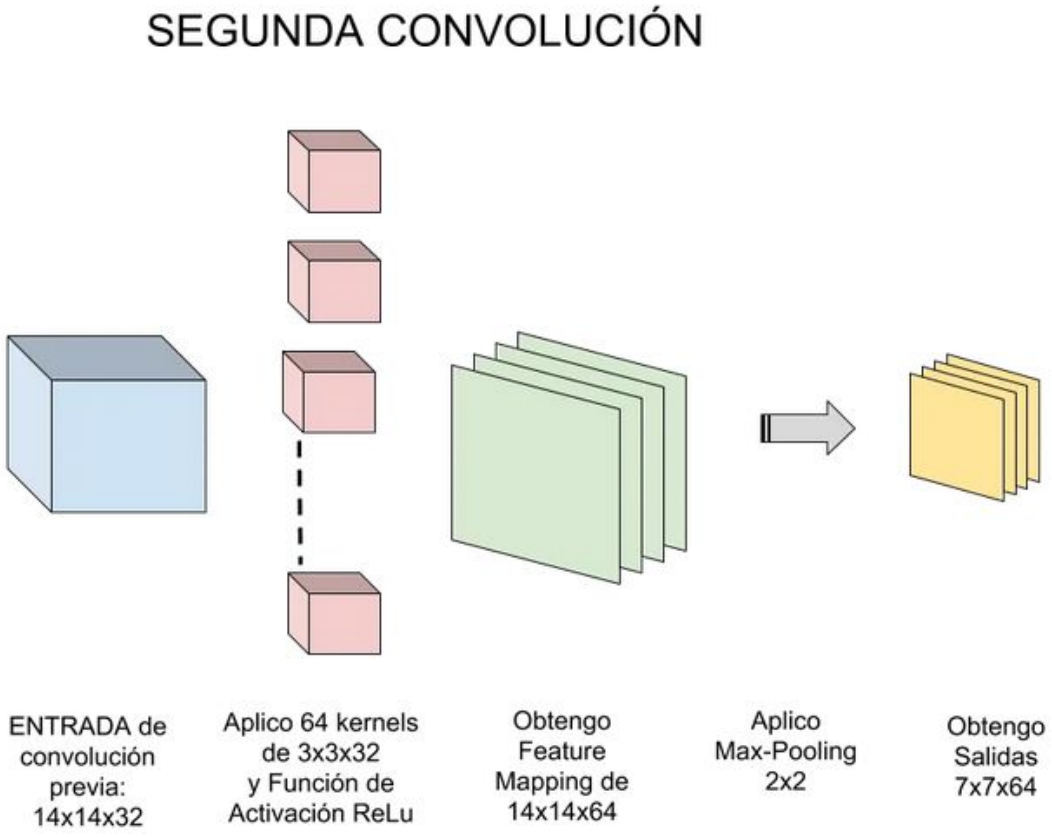

imagen 11 - ilustración segunda convolución

La 3er convolución comenzará en tamaño 7×7 pixels y luego del max-pooling quedará en 3×3 con lo cual podríamos hacer sólo 1 convolución más. Llegada a la última convolución, solo queda conectar esta última a red neuronal tradicional.

Para terminar, tomaremos la última capa oculta a la que hicimos subsampling, que se dice que es «tridimensional» por tomar la forma 3x3x128 y la aplanamos (esto es que deja de ser tridimensional, y pasa a ser una capa de neuronas tradicionales).

Entonces, a esta nueva capa oculta tradicional, le aplicamos una función Softmax que conecta contra la capa de salida final que tendrá la cantidad de neuronas correspondientes con las clases que estamos clasificando.

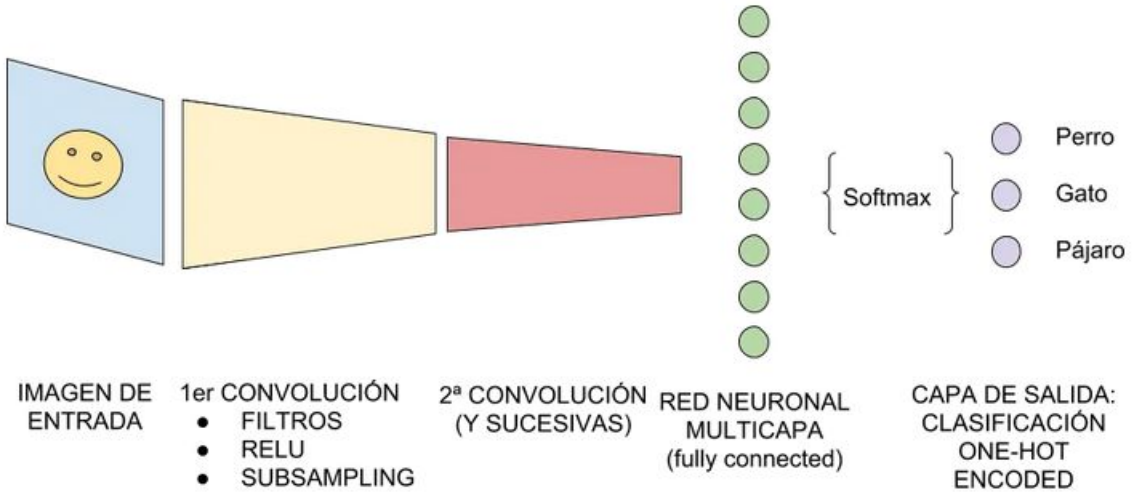

imagen 12 - ilustración CNN proceso entero

4 - YOLO

Cuando a alguien se le menciona por primera vez el algoritmo YOLO lo primero que uno dirá es que es sumamente rápido. Pero, ¿A qué se debe esta velocidad?

YOLO, a diferencia de otros algoritmos tradicionales del mismo campo (R-CNN y sus familiares) es considerado un algoritmo de una fase, mientras el conjunto R-CNN, Fast R-CNN, Faster R-CNN son considerados algoritmos de 2 fases. Otro algoritmo de una fase podría ser, por ejemplo, SSD (Single Shot Detector). Estas fases hacen referencia al trabajo que realiza el algoritmo. R-CNN en su primer fase tomará una imagen como entrada y “propondrá” regiones de interés, las cuales se supone que contendrán o hay grandes chances de que contengan un objeto. En su segunda fase se toma cada una de estas regiones propuestas y son procesadas a través de la CNN. Este proceso es extremadamente costoso (pero a la vez, preciso). Es por esto que R-CNN y sus derivados son lentos. YOLO, que recordemos que es de una única fase, en cambio toma como entrada toda la imagen y nos dará como salida un tensor, el cual contendrá datos que son necesarios para “graficar” los límites de las bounding boxes (o cajas delimitadoras) y a su vez contendrá datos sobre las probabilidades de las clases que se encuentran en dichas cajas.

Funcionamiento de YOLO

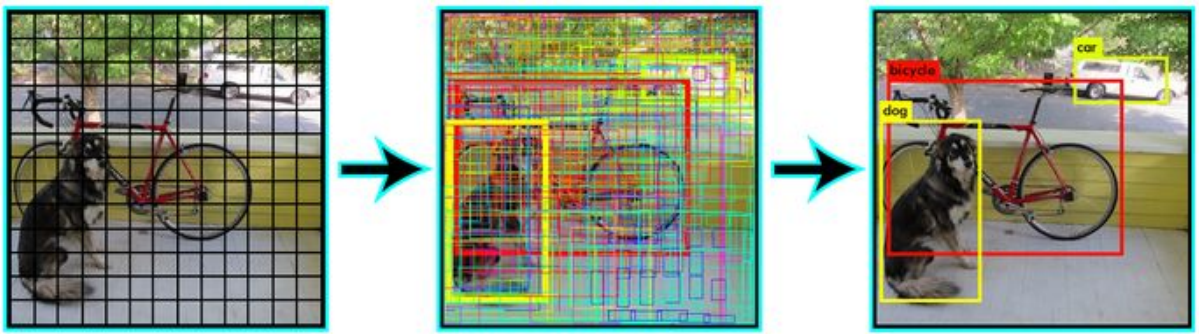

imagen 13 - proceso YOLO resumido

Los algoritmos de una fase encaran el problema de la detección como si fuera una regresión, en vez de clasificación. En vez de seleccionar partes de interés de una imagen, predicen clases y bounding boxes para toda una imagen.

Para entender el algoritmo YOLO es indispensable entender qué es lo que se está prediciendo. Se apunta a predecir la clase de un objeto y su bounding box, especificando su localización en la imagen.

Cada bounding box puede ser detallada de la siguiente forma con **5** características:

1. Centro de la bounding box (Bx, By)
2. Ancho (Bw)
3. Altura (Bh)
4. Pc, que es la probabilidad de que haya un objeto en la bounding box. Se la puede llamar también "*confidence score*".

A su vez, la bounding box tendrá una serie de valores extra, el o los valores C, el cual corresponde a una clase de un objeto. (Estos valores son utilizados para la codificación *one hot encoding*, la cual básicamente es una forma *hard-codeada* de detallar en nuestro caso a qué clase pertenece nuestro objeto. Ej, si se tienen 3 clases y nuestro objeto pertenece a la clase número 2, entonces C=0,1,0)

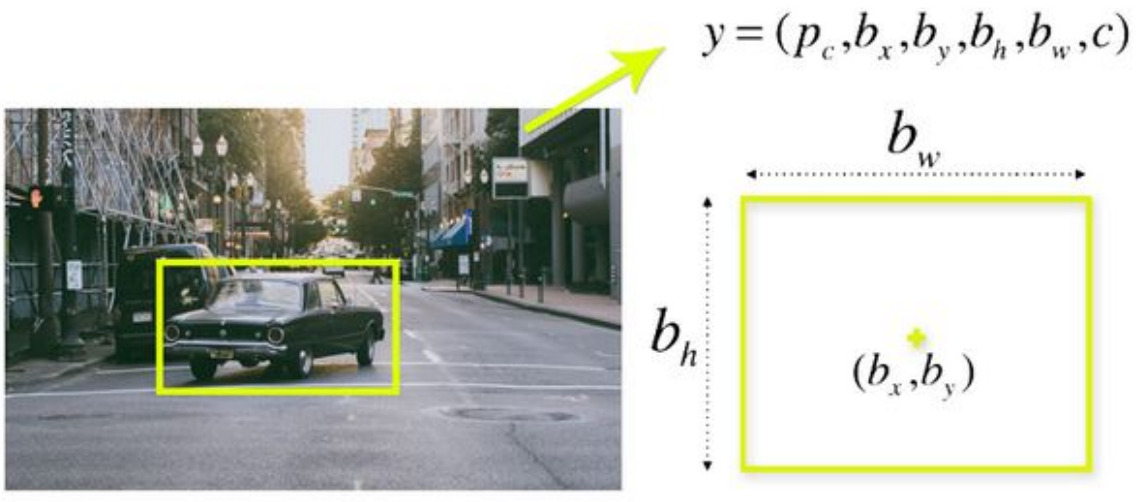

imagen 14 - bounding box explicación

La forma que tienen los valores dentro de una bounding box para C=80es la siguiente

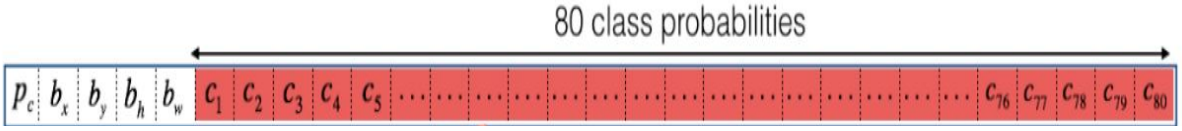

imagen 15 - valores dentro de la bounding box

Ahora que se entiende lo que contiene una bounding box en nuestro algoritmo YOLO, se procederá a explicar su funcionamiento a partir de una entrada (imagen).

Lo primero que hace yolo al recibir una imagen, es dividir la imagen en celdas contenidas dentro de una grilla de dimensiones SxS (típicamente 19x19 o 13x13).

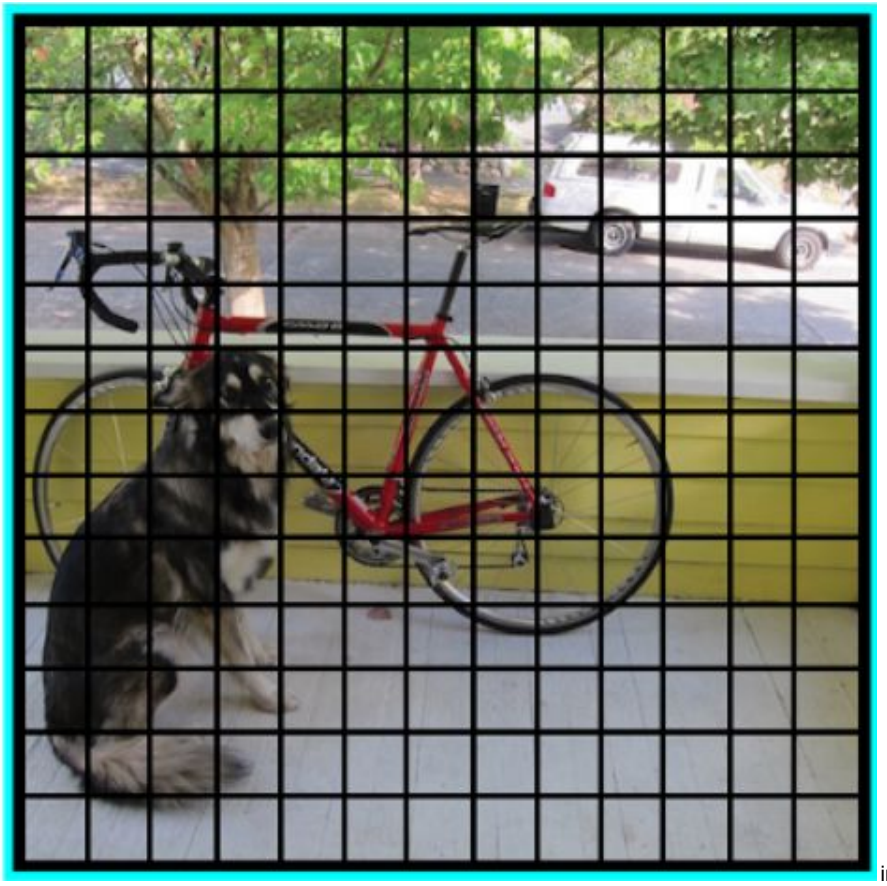
imagen 16 - grilla inicial SxS

Luego, cada celda es responsable de obtener una cantidad B de bounding boxes, usualmente siendo B=5 (en YOLOv3 B suele ser igual a 3). Claro que la bounding box puede ser más grande que una celda si el posible objeto es más grande que la celda en sí.

Hasta el momento, se tiene SxSxB posibles bounding boxes, aunque la mayoría tienen una probabilidad asociada de detección sumamente baja.

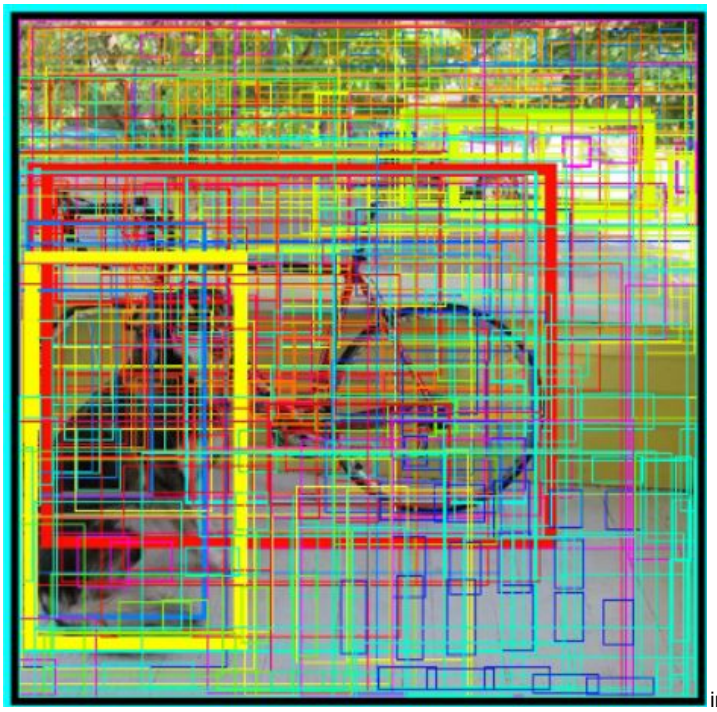
imagen 17 - bounding boxes totales

Luego de tener todas las bounding boxes posibles, se empiezan a descartar la gran mayoría. Los criterios para descartar son 2:

A. Que el valor Pc que contiene la probabilidad de que haya un objeto sea bajo.
B. Que varios bounding boxes esten conteniendo el mismo objeto (Non-maxima suppression).

Una vez que se descarta la mayoría de las bounding boxes, solo quedan unas pocas, las cuales se presumen que contendrán una probabilidad alta de que hayan encontrado un objeto y una clase asociada.

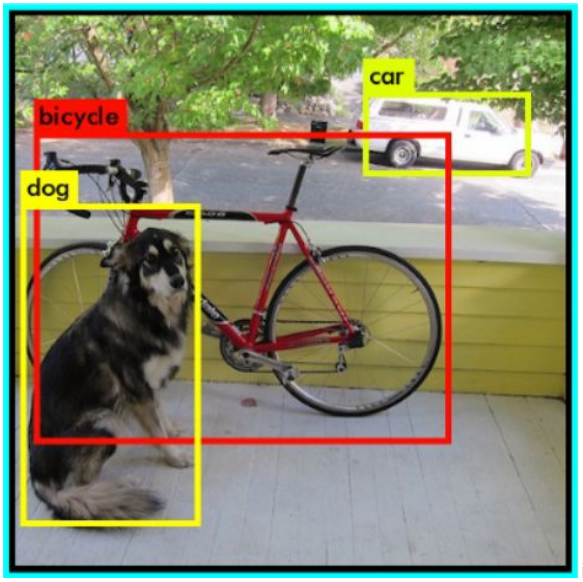

imagen 18 - bounding boxes finales

Si bien la imagen que visualizamos al final de todo el proceso es la imagen de entrada con bounding boxes y clases asociadas, lo que realmente nos da como salida el algoritmo es un tensor de SxS.(B.(5+C)), donde

SxS es la dimensión de la grilla inicial

B la cantidad de bounding boxes por celda de la grilla

5+C son los datos de las bounding boxes + el one hot encoding de las clases que tolera el modelo.

Estructura del algoritmo

La estructura del algoritmo YOLO difiere bastante entre una versión y otra. Su primer versión contaba con 24 capas convolucionales.

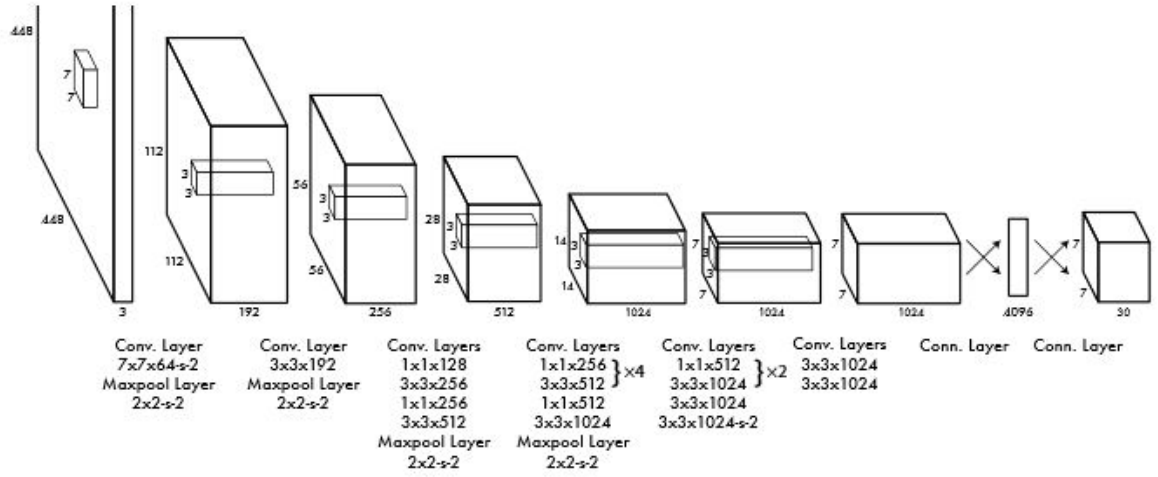

imagen 19 - estructura YOLO primer versión

Mientras que YOLOv3 utiliza solo capas convolucionales, por lo que es una red totalmente convolucional (FCN) y contiene 53 capas convolucionales, cada una de ellas seguida de la capa de normalización por lotes y la activación Leaky ReLU. No se usa ninguna forma de agrupación, y se usa una capa convolucional con paso 2 para reducir la muestra de los mapas de características. Esto ayuda a prevenir la pérdida de características de bajo nivel que a menudo se atribuyen a la agrupación.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

imagen 20 - estructura yoloV3

Cada bounding box predice las clases que puede contener, utilizando la clasificación de múltiples capas. A diferencia de su primer versión, YOLOv3 no usa más softmax ya que sus autores encontraron que es innecesario para un buen rendimiento, en su lugar, simplemente usan clasificadores logísticos independientes. Durante el entrenamiento, utilizaron la pérdida de entropía cruzada binaria para las predicciones de clase. Esta formulación ayuda cuando el algoritmo es utilizado en dominios más complejos como el *Open Images Dataset*. En este conjunto de datos hay muchas etiquetas (*labels*) superpuestas (es decir, Mujer y Persona). Usar un softmax impone la suposición de que cada cuadro tiene exactamente una clase, que a menudo no es el caso. Un enfoque *multilabel* modela mejor los datos y por ende los resultados, de ahí la mejora en precisión de YOLO a YOLOv3.

5-YOLO vs otros

Antes de iniciar esta comparación, se debe aclarar que en el mundo de detección de objetos a través de imágenes, hay muchas métricas y formas de medirlas. Existen datasets de imágenes cuyo único propósito es medir la performance de estos algoritmos. Hay formas de métricas donde se miden los mAPs, AP, AP50, AP75 (en estos se entiende que .50 y .75 son los valores que deben alcanzar los algoritmos respecto del valor de IoU.). A la vez, hay puntajes sobre la performance de estos algoritmos mientras se varía el tamaño de las imágenes. También existen tests de performance que se miden únicamente en el entrenamiento, haciendo que se deban variar los algoritmos internos de obtención de *features* (usualmente estos algoritmos internos son llamados *backbone*).

Dicho esto, se presentan comparaciones generales entre yolo, faster r-cnn (algoritmo de 2 fases) y SSD (algoritmo de una fase, al igual que yolo).

Frames por segundo (FPS)

En este gráfico se puede visualizar la diferencia notable que hay entre las velocidades de los algoritmos. *Nota: "High" representa el máximo alcanzado y "Low" el mínimo alcanzado*.

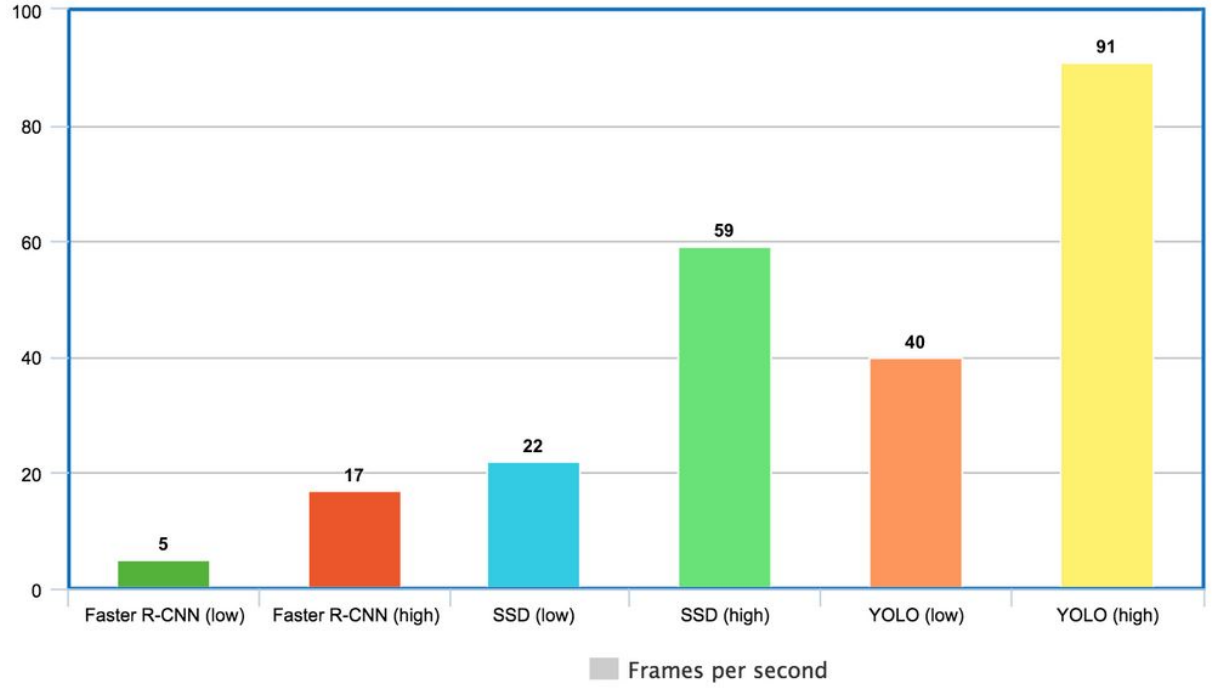

imagen 21 - comparación de algoritmos FPS

Precisión mAP

En este gráfico se podrá visualizar la diferencia entre precisión. Esta prueba se llevo a cabo utilizando el dataset COCO. Este dataset es muy popular en este campo y resulta ser un reto para estos algoritmos. Usualmente los algoritmos logran puntajes "bajos" de mAP si se los corre con este dataset.

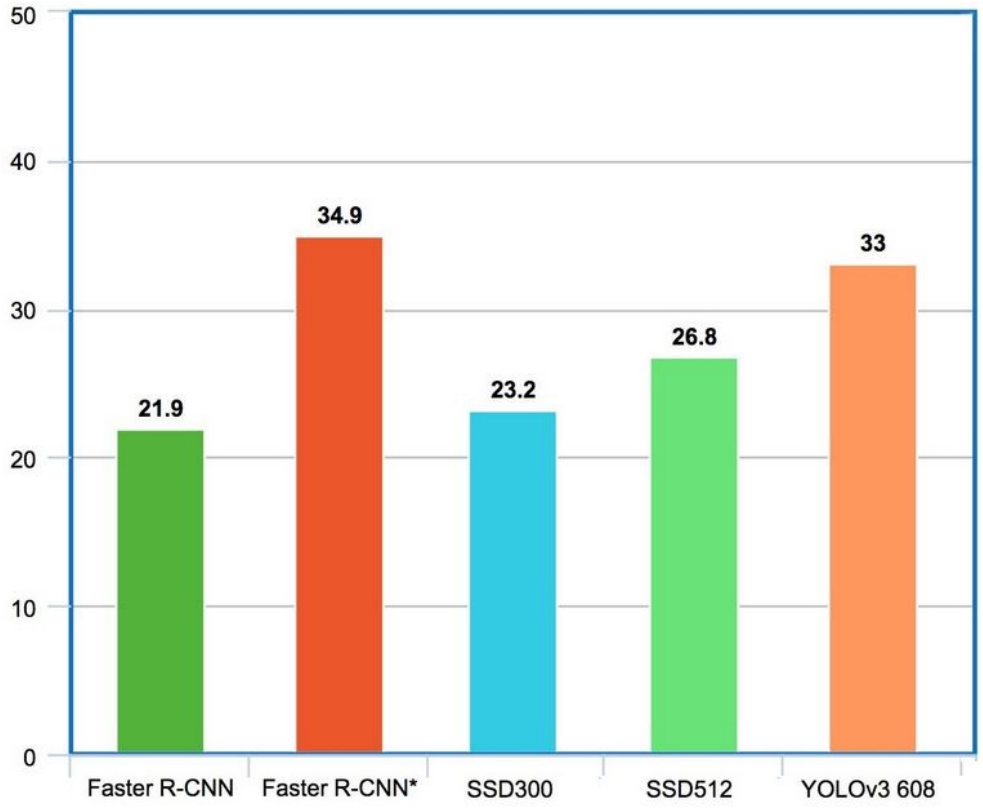

imagen 22 - comparación algoritmos - AP

6-Limitaciones de yolo

-YOLO funciona mal o tiene una mala performance en objetos pequeños en sus versiones anteriores a la YOLOv3, particularmente aquellos que aparecen en grupos, ya que cada celda de la grilla predice solo B = 3 cuadros delimitadores de la misma clase, creando una restricción espacial que limita el número de objetos predichos por celda de la grilla. Aunque en la versión YOLOv3 es al revés, se soluciona la performance agravante que tenía para objetos pequeños pero pierde precisión para objetos grandes.

7-Implementación

7.1-Herramientas

7.1.1-Dataset coco

COCO es un dataset de detección, segmentación y capturaciones de objetos a gran escala. COCO tiene varias características:

-Segmentación de objetos
-Reconocimiento en contexto
-Segmentación de superpíxeles

-330K imágenes (> 200K etiquetadas)
-1,5 millones de instancias de objetos
-80 categorías de objetos (es decir, 80 clases)
-91 categorías de cosas

7.1.2-Framework darknet

Darknet es un framework de redes neuronales de código abierto, escrito en C (uno de los factores sobre porque al final del dia yolo es rápido..). Es rápido y es sencillo de instalar. Tiene soporte para procesamiento por medio de GPU.

Darknet es utilizado dado que en su interior cuenta con la configuración propia de las redes neuronales convolucionales de YOLO.

7.1.3-OpenCV

OpenCV es una librería de código abierto de “visión computacional”. Si bien esta librería contiene muchos plugins para otras librerías conocidas como Tensorflow, en este caso será utilizada para consumir el framework darknet, el cual se encuentra dentro de OpenCV desde su versión 3.4.2+

7.2-Código

A continuación, se presenta el código.

Cabe destacar, que la implementación de yolo que se mostrara, al estar hecho en openCV, podemos pasarle pesos previamente entrenados. Esto simplifica en gran manera todo el algoritmo y se logra una abstracción considerable.

Empezamos con las librerías

```
#Librerias
import numpy as np
import time
import cv2
import os
```

Declaramos la ruta de la imagen que queremos procesar a través del algoritmo.

```
#path de la imagen en la cual se detectaran y clasificaran objetos
path_imagen="/home/ima/Desktop/python/practica/yolo/yolo3/imagen4.jpeg"
```

Se cargan los labels de las clases del dataset COCO en los que el modelo YOLO fue entrenado previamente.

```
#carga los labels de las clases del dataset COCO en los que el modelo YOLO fue entrenado
labelsPath = os.path.sep.join(["/home/ima/Desktop/python/practica/yolo/yolo3/yolo-coco", "coco.names"])
LABELS = open(labelsPath).read().strip().split("\n")
```

Se le agregan colores a los labels para ser mostrados en la imagen de salida.

```
# se le añaden colores de forma aleatoria a los labels
np.random.seed(42)
COLORS = np.random.randint(0, 255, size=(len(LABELS), 3),
    dtype="uint8")
```

Declaramos donde están los pesos y la configuración de la red neuronal interna.

```
# paths de donde estan guardados los pesos de la red neuronal y la configuracion de esta.
weightsPath = os.path.sep.join(["/home/ima/Desktop/python/practica/yolo/yolo3/yolo-coco", "yolov3.weights"])
configPath = os.path.sep.join(["/home/ima/Desktop/python/practica/yolo/yolo3/yolo-coco", "yolov3.cfg"])
```

A través de openCV se carga el framework de darknet y se genera un objeto a partir de este.

```
#A traves de openCV, se carga el framework de darknet y se genera el objeto de este.
print("[INFO] loading YOLO from disk...")
net = cv2.dnn.readNetFromDarknet(configPath, weightsPath)
```

Se carga la imagen de entrada y se obtienen sus dimensiones.

```
# se carga la imagen y se obtienen las dimensiones de esta
image = cv2.imread(path_imagen)
(H, W) = image.shape[:2]
```

Se obtienen los nombres de la capa de salida. Esto es un requerimiento por parte del framework Darknet.

```
#Se determinan los nombres de la capa de salida. Esto es requerido para el funcionamiento del framework Darknet
ln = net.getLayerNames()
ln = [ln[i[0] - 1] for i in net.getUnconnectedOutLayers()]
```

Se hace un “*blob*” sobre la imagen de entrada. Esto básicamente es un preprocesamiento sobre la imagen. Blob permite el re-escalado de la imagen, intercambiar valores RGB, redimensionar la imagen, etc. Luego del preprocesamiento de la imagen, esta pasa a través de la red

```
# Se construye un "blob" de la imagen de entrada.
# Se pasa la imagen a traves de la red
blob = cv2.dnn.blobFromImage(image, 1 / 255.0, (416, 416),
    swapRB=True, crop=False)
net.setInput(blob)
start = time.time()
layerOutputs = net.forward(ln)
end = time.time()

# Se muestra el tiempo que le llevo a YOLO
print("[INFO] YOLO took {:.6f} seconds".format(end - start))
```

Se crean las listas que contendrán los datos sobre los bounding boxes, confidence scores y IDs de clases.

```
#Se inicializan las listas que contendran las coordenadas de las bounding boxes, confidence scores y
# los IDs de las clases

boxes = []
confidences = []
classIDs = []
```

Se hace un loop a través de cada capa de salida, y para cada cada se hace un loop sobre cada detección, en el cual se extraen los datos sobre IDs de clases, y confidence score sobre el objeto detectado. Luego se filtran las predicciones débiles, y se escalan las bounding boxes a las proporciones de la imagen. Por último se agregan los datos recolectados a las listas creadas previamente.

```
# loop sobre cada capa de salida
for output in layerOutputs:
    # loop sobre cada deteccion
    for detection in output:
        # se extrae la clase, ID y confidence score del objeto detectado

        scores = detection[5:]
        classID = np.argmax(scores)
        confidence = scores[classID]

        # se filtran las predicciones debiles

        if confidence > 0.5:
            # se escalan las bounding boxes a las proporciones de la imagen
            box = detection[0:4] * np.array([W, H, W, H])
            (centerX, centerY, width, height) = box.astype("int")

            x = int(centerX - (width / 2))
            y = int(centerY - (height / 2))

            # se actualiza la lista de bounding boxes, confidence scores y IDs de clases
            boxes.append([x, y, int(width), int(height)])
            confidences.append(float(confidence))
            classIDs.append(classID)
```

Se aplica NMS.

```
# se aplica NMS para suprimir bounding boxes debiles
idxs = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.3)
```

Nos aseguramos de que exista una detección.

```
# nos aseguramos que exista una deteccion
if len(idxs) > 0:
    # loop sobre los indices
    for i in idxs.flatten():
        # se extraen las coordinadas de las bounding boxes
        (x, y) = (boxes[i][0], boxes[i][1])
        (w, h) = (boxes[i][2], boxes[i][3])

        # se dibuja la bounding box
        color = [int(c) for c in COLORS[classIDs[i]]]
        cv2.rectangle(image, (x, y), (x + w, y + h), color, 2)
        text = "{}: {:.4f}".format(LABELS[classIDs[i]], confidences[i])
        cv2.putText(image, text, (x, y - 5), cv2.FONT_HERSHEY_SIMPLEX,
            0.5, color, 2)
```

Se muestra la imagen de salida con las bounding boxes y clases correspondientes.

```
# se muestra la imagen de salida
cv2.imshow("Image", image)
cv2.waitKey(0)
```

7.3-Ejemplos (buenos y malos)

7.3.1 - Ejemplos positivos

a)

imagen 23 - ejemplo positivo 1

En este ejemplo, reconoce a todas las personas, 2 autos, y un semáforo. Nótese que la probabilidad del semáforo es un poco baja, 51%.

b)

imagen 24 - ejemplo positivo 2

Se logran obtener varias clases correctamente, aunque la clase “zanahoria” no se distingue si es correcta del todo.

c)

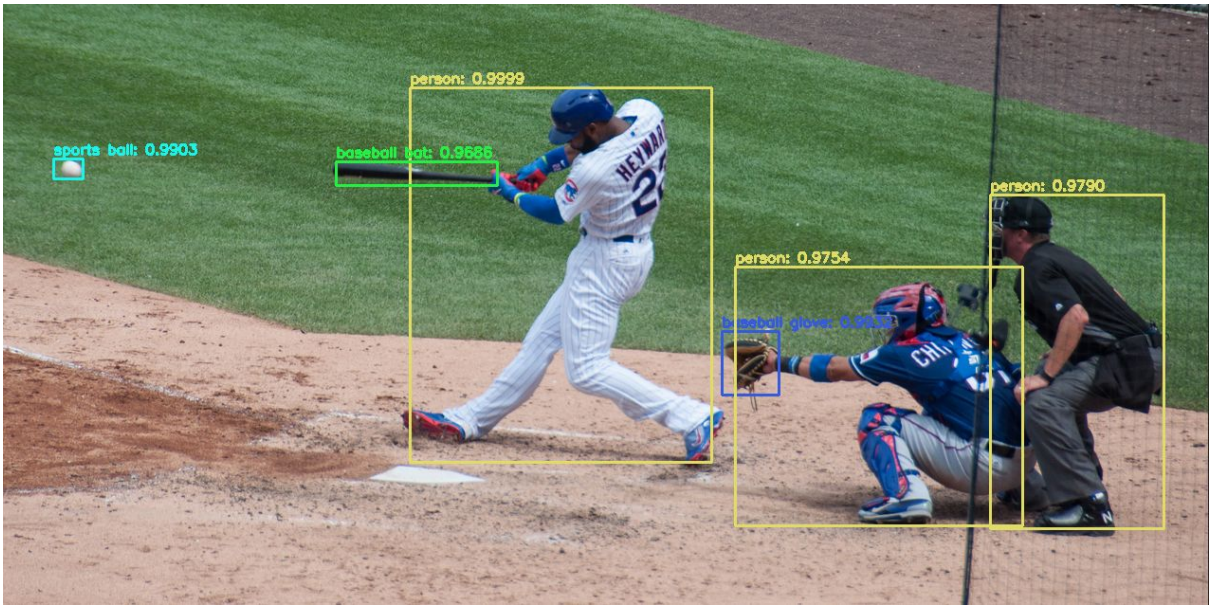

imagen 25 - ejemplo positivo 3

Como podemos ver, también detecta objetos de deportes, como los guantes y bate de baseball.

7.3.2 - Ejemplos negativos

a)

imagen 26 - ejemplo negativo 1

Como podemos ver, primero que no detecta todos los animales en la imagen. Esto se puede dar por varios motivos. El principal, es que algunos de ellos no se encuentren en el dataset. Otro motivo puede ser que simplemente el factor de confianza no sea lo suficientemente bueno como para catalogar a estos animales faltantes. Segundo, algunos animales tienen una clase asociada errónea. Nuevamente, esto puede ser por la falta de la clase correcta en el dataset, por lo que el algoritmo intenta asociar el animal con la clase más “parecida”, como en el caso del rinoceronte. Por otro lado, en la imagen hay un elefante pero este fue catalogado como un ave, con un 54% de seguridad. El label elefante existe en el dataset coco, así que es difícil entender el por qué falló en su clasificación.

b)

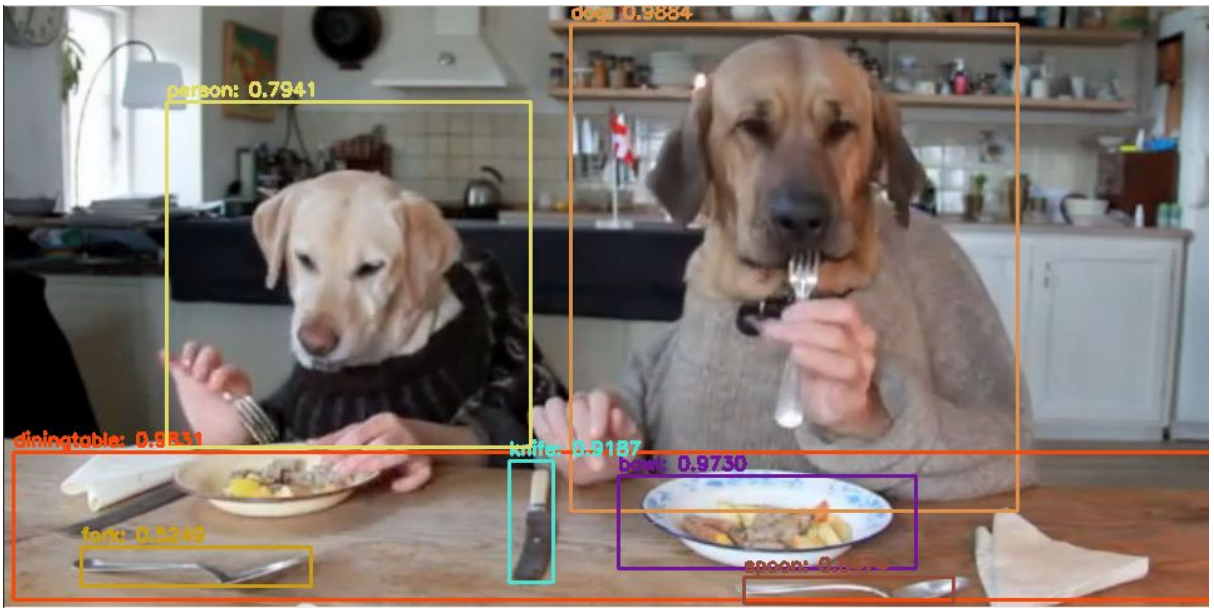

imagen 27 - ejemplo negativo 2

Esta imagen puede resultar un poco tramposa, dado que, si bien los objetos sobre la mesa son normales, los perros están acompañados de un cuerpo y extremidades. Es entendible que se confunda el algoritmo en este caso.

8-Conclusiones

El algoritmo yolo cumple con las expectativas. El proceso de introducir una imagen y obtener una salida apenas toma un segundo (CPU), aunque es cierto que no es muy certero, como se puede ver en el caso de los ejemplos negativos. No obstante, sigue siendo uno de los mejores algoritmos para la deteccion y clasificacion de objetos de tiempo real. Siempre que se necesite esta detección en tiempo real recomendaría el uso de YOLO.

9-Mejoras

Hablar de mejoras en este sistema tan complejo puede ser un poco ambicioso. Si bien en este trabajo se habla de YOLO en general, haciendo hincapié en YOLOv3, al momento de escribirlo se encuentra en existencia yolov4 y yolov5, los cuales fueron publicados en Abril y Junio de 2020. Cabe destacar que el creador original de yolo, Joseph Redmon, luego de la versión V3 decidió no seguir con él, dado que yolo podría ser utilizado con fines poco éticos o incluso militares. Como estas versiones son más recientes, se entiende que son mejoras.

YOLOv4 tiene un aumento del 10% en AP (*Average Precision*) y un aumento del 12% en FPS (*frames per second*) si se lo compara con YOLOv3.
En cambio, para YOLOv5, aún no hay comparaciones directas con yoloV4, por lo que solo podemos “creer” que es mejor, como afirma su autor, Glenn Jocher.

10-Bibliografía

-https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/
-https://pjreddie.com/media/files/papers/YOLOv3.pdf
-https://medium.com/analytics-vidhya/yolo-v3-theory-explained-33100f6d193
-https://arxiv.org/pdf/1506.02640.pdf
-https://towardsdatascience.com/conv-nets-for-dummies-a-bottom-up-approach-c1b754fb14d6
-https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/
-https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/
-https://cv-tricks.com/how-to/running-deep-learning-models-in-opencv/
-https://medium.com/@enriqueav/detecci%C3%B3n-de-objetos-con-yolo-implementaciones-y-como-usarlas-c73ca2489246
-https://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/
-https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c
-https://github.com/pjreddie/darknet/blob/master/cfg/darknet53.cfg
-https://cocodataset.org/#home
-https://appsilon.com/object-detection-yolo-algorithm/
-https://manalelaidouni.github.io/manalelaidouni.github.io/Understanding%20YOLO%20and%20YOLOv2.html
-https://pjreddie.com/darknet/
-https://towardsdatascience.com/whats-new-in-yolov4-323364bb3ad3
-https://tech.amikelive.com/node-718/what-object-categories-labels-are-in-coco-dataset/