Detección de caras usando una red YOLO preentrenada

Visión por Computador

Miguel Lentisco Ballesteros Antonio Coín Castro

Curso 2019-20

Índice

1. Introducción

El objetivo de este proyecto, es usar la red neuronal YOLOv3 preentrenada en la base de datos COCO, para detectar caras sobre la base de datos WIDERFACE.

Veremos el funcionamiento y estructura general de YOLOv3, como la hemos usado y entrenado para detectar, resultados y las conclusiones.

2. YOLOv3

2.1. Descripción

YOLOv3 ("You only look once" versión 3) es una red neuronal con arquitectura **completamente convolucional** dirigida a detección de objetos, que destaca como uno de los algoritmos de detección más rápidos que hay; si bien es cierto que hay otros con mejor tasa de precisión, YOLO nos da la ventaja en su bajo tiempo de ejecución frente a los otros algoritmos, lo cual es esencial cuando necesitamos hacer reconocimiento de objetos en **tiempo real**.

2.2. Funcionamiento general

YOLO realiza detección en 3 escalas distintas, de manera que devuelve un tensor3D para cada escala del mismo tamaño que la escala en la que está detectando, codificando la información de cada celda: las coordenadas de la caja, la puntuación de si es un objeto (querremos que sea 1 en el centro de la bounding box y 0 en caso contrario) y puntuación de cada clase. Además, en cada escala se predicen 3 cajas de tamaño prefijado (**anchor**), por lo tanto se tiene que devuelve un tensor3D de tamaño NxNx[3x(4+1+M)], con N el tamaño de la escala y M el nº de clases a detectar.

El entrenamiento se encarga de aprender la mejor caja (la que se superponga más sobre el ground truth) y de ajustar las coordenadas para la caja escogida y para obtener el tamaño de las cajas prefijadas se calcula usando un método de clustering K-medias al dataset antes de entrenar; este diseño permite que la red aprenda mejor y más rápido las coordenadas de las bounding box.

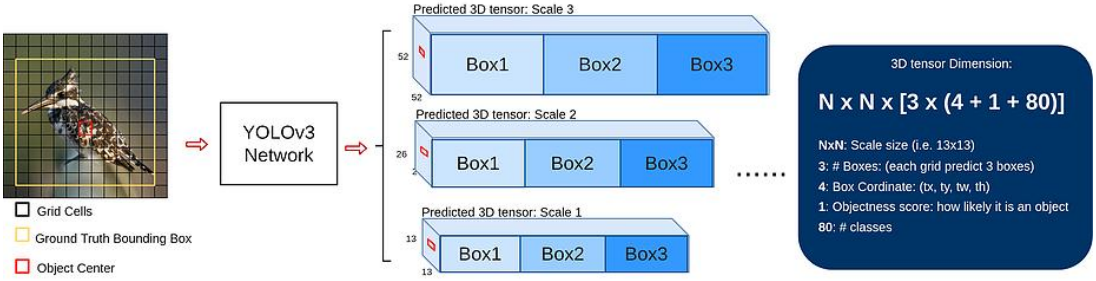


Figura 1: Funcionamiento general

2.3. Arquitectura

Como ya hemos comentado, YOLO usa una arquitectura completamente convolucional (permitiendo que podamos pasar cualquier tamaño de imagen), con 75 capas convolucionales en total.

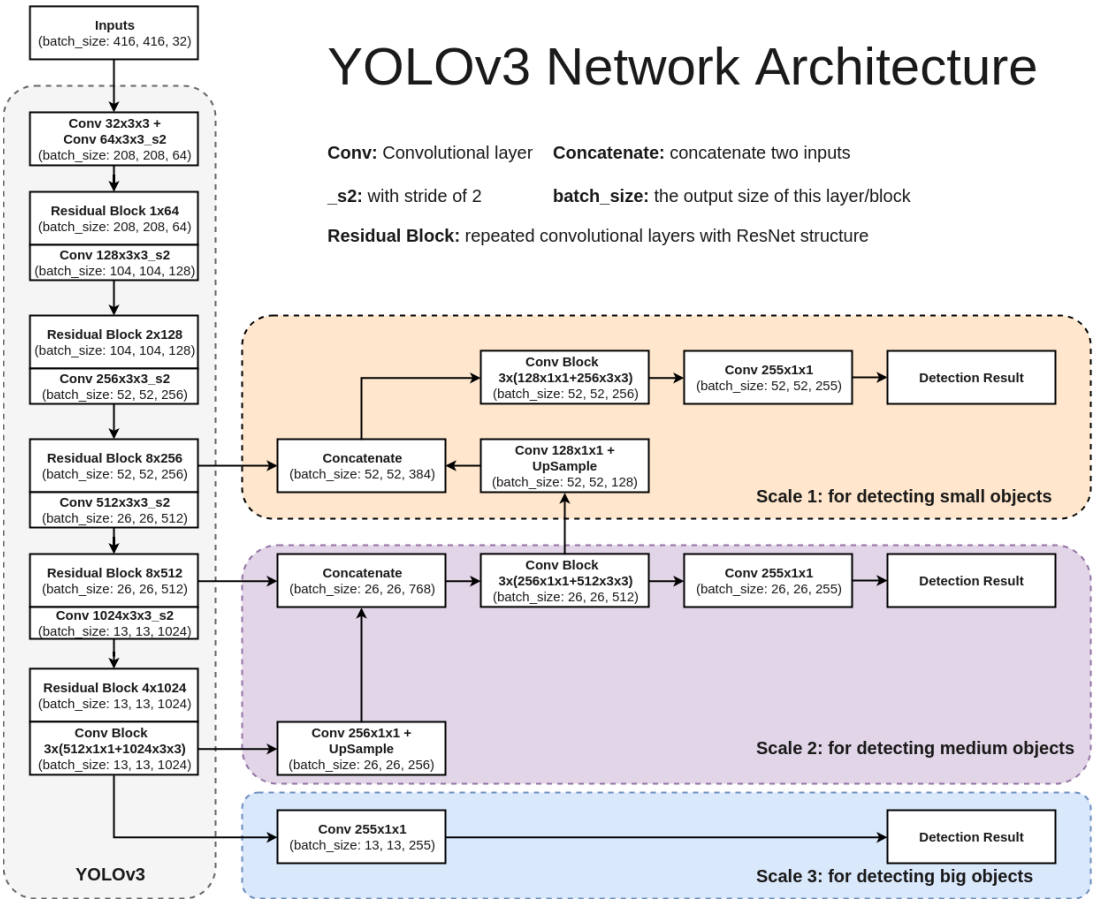


Figura 2: Arquitectura de YOLOv3

El modelo está comprendido en dos partes:

- **Darknet-53**: es el extractor de características a distintas escalas, que se compone principalmente por 52 capas convolucionales, que incluye bloques residuales (2 convoluciones + 1 skip), y con capas convolucionales con stride 2 antes de cada bloque para hacer downsampling sin necesidad de usar pooling. Además después de cada convolucional se añade una capa BatchNormalization y con activación Leaky ReLU.

  Vemos como se van incluyendo pequeños bloques con tamaño de filtros pequeño, y aumentamos ambos valores conforme profundizamos la red. Al final de cada bloque 8x (capa 36 y 61) se pasará una conexión a las escalas pequeña y mediana (lo veremos después).

- **Detección en escalas**: como los objetos a detectar pueden aparecer de distintos tamaños y queremos detectarlos todos, tenemos un problema puesto que la

	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			

Figura 3: Darknet-53

red conforme es más profunda más le cuesta detectar objetos pequeños. YOLO resuelve esto usando una estructura de detección piramidal (Feature Pyuramid Network) que se encarga de detectar en 3 escalas distintas (pequeño, mediano y grande).

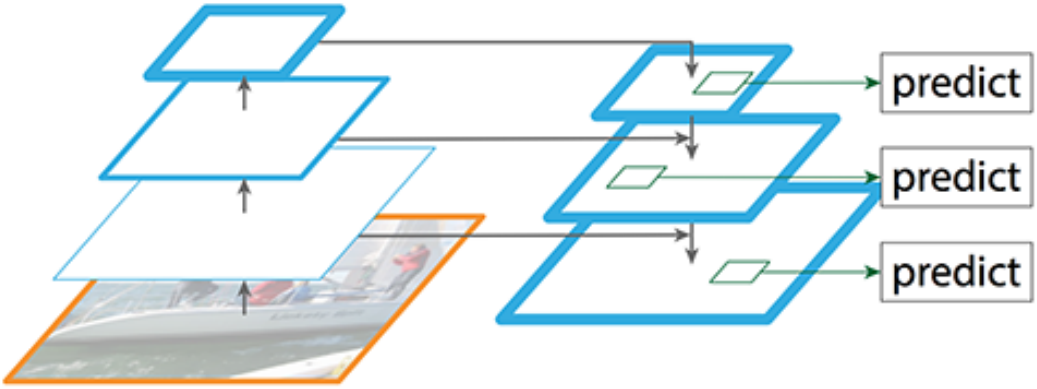
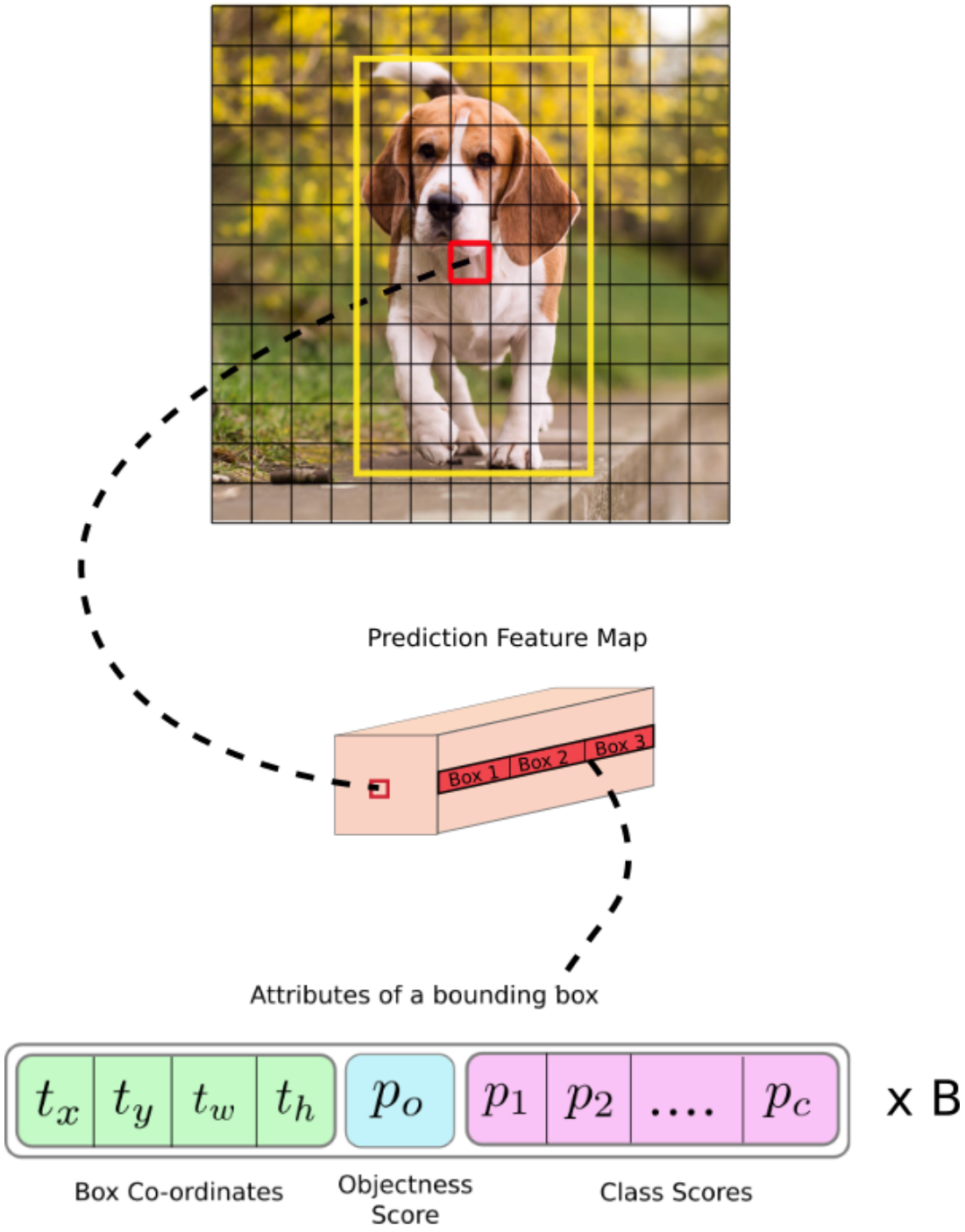


Figura 4: Feature Pyramid Network

Tomando el mapa de características que produce **Darknet** final se pasa a la escala grande directamente y a la mediana con upsampling x2; en la grande se pasa al detector, y en la mediana se concatena con otro mapa de características (capa 61) menos profundo que se pasa a la escala mediana con upsampling x2 y a la mediana directamente al detector. Finalmente repetimos el proceso para la escala pequeña usando otro mapa de características (capa 36) menos profundo todavía concatenado con lo anterior que se pasa a un detector.

2.4. Predicción

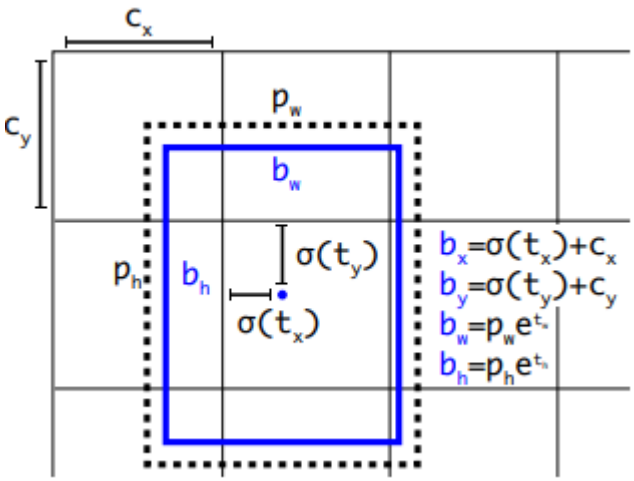
Veamos lo que produzca en la capa de detección en cada escala, que consiste en una serie de valores (coordenas de la caja, puntuación de objeto, y puntuación de clase) por cada una de las 3 cajas prefijadas.



2.4.1. Caja

Se predicen 4 coordenadas para cada bounding box, las coordenadas x e y del centro, y la anchura y altura de la caja, denotémoslas t_x, t_y, t_w, t_h. Si la celda está desplazada de la esquina superior izquierda por un (c_x, c_y), y siendo p_w, p_h la anchura y altura de la caja prefijada, y σ una función sigmoide, entonces las coordenadas de la caja predecidas son:

$$b_x = \sigma(t_x) + c_x$$
$$b_y = \sigma(t_y) + c_y$$
$$b_w = p_w e^{t_w}$$
$$b_h = p_h e^{t_h}$$



Aunque en principio podría detectarse directamente las coordenadas, al entrenar ocasiona muchos gradientes inestables, por lo que se funciona mucho mejor prefijando una caja y aplicando transformaciones logarítmicas; en nuestro caso al tener 3 cajas fijadas obtendremos 4 coordenadas por cada caja. Para calcular las coordenadas se usa como función de perdida la suma de los errores cuadrados.

Realmente estas coordenadas no son absolutas, puesto que son relativas a la esquina superior izquierda de la imagen, y además se normalizan entre la dimensión de la celda del mapa de características; por tanto si las coordenadas del centro predichas son mayores que 1 producen que se salga del centro, de ahi que usemos la función sigmoide (deja entre 0 y 1). a la altura y anchura les pasa igual, y son normalizadas por la anchura/altura de la imagen.

2.4.2. Objeto

La puntuación de objeto consiste en como de probable es que un objeto esté dentro de la caja, por lo que idealmente queremos es que la celda del centro de la caja sea cercana a 1, mientras que por las las zonas exteriores cercanas a la caja sea casi 0.

2.4.3. Clase

Cada caja predice la clase que puede tener el bounding box mediante clasificación multietiqueta, no usando softmax puesto que no influye en términos de rendimiento, pero de esta manera podemos etiquetar con varias etiquetas; así, se usa binary cross-entropy loss durante el entrenamiento.

2.5. Detección

Cuando hacemos detección obtendremos muchas cajas, por lo que tendremos que filtrar. Primero ordenamos las cajas según su puntuación de objeto, ignoramos las que no sobrepasen un cierto umbral (por ejemplo 0.5) y finalmente aplicaremos supresión de no-máximos para condensar muchas cajas que estén casi superpuestas.

2.6. Evaluación

3. Cosas por hacer

Varios:

- Entender todo el código. Eliminar lo que no sea necesario.
- Adaptar código para poder evaluar el conjunto de test (a partir de filelist, sin anotaciones).
- Ver por qué no coincide la métrica de evaluación de `evaluate_coco` con la de `codalab`. Reimplementar para que coincidan. Posiblemente cambiar el cálculo de AP a la interpolación en 101 pasos (https://kharshit.github.io/blog/2019/09/20/evaluation-metrics-for-object-detection-and-segmentation). Si no funciona, probar con 11 pasos.

Entrenamiento:

- Hacer finetuning a partir de los pesos de COCO iniciales (congelar unas cuantas capas, ¿cuáles?)
- Cambiar optimizador a SGD, RMSProp, Adabound(https://github.com/Luolc/AdaBound/blob/maste
  <—
- Entrenar más épocas. <—
- Entrenar con un valor mayor de *xywh_scale* en el config. Por ejemplo 2? <—
- Entrenar con un mayor tamaño de entrada de las imágenes. Ahora mismo en Colab no es viable.
- Aumentar el umbral *ignore_thresh*, por ejemplo a 0.6 ó 0.7.

Evaluación:

- Aumentar tamaño de entrada (no sé si tiene sentido que supere al input_size de entrenamiento) <—
- Aumentar umbral supresión de no máximos, por ejemplo a 0.6

Opcionales:

- Reimplementar la funcíón de supresión de no máximos en su versión vectorizada, para que sea más rápida. Adaptar implementación de https://www.pyimagesearch.com/2015/02/16/faster-non-maximum-suppression-python/

4. Información a tener en cuenta

- The output of the model is, in fact, encoded candidate bounding boxes from three different grid sizes: 13x13, 26x26 y 52x52.
- Explicación de mAP: `https://medium.com/@jonathan_hui/map-mean-average-precision-for-ob`

PASOS SEGUIDOS:

1. Convertir las anotaciones de WIDERFACE a formato VOC. Para ello se ha usado el archivo `convert.py`, adaptado de https://github.com/akofman/wider-face-pascal-voc-annotations/blob/master/convert.py
2. Generar anchor boxes para nuestro conjunto usando k-means con gen-anchors.py, y ponerlos en el config.
3. Descargar los pesos backend.h5 preentrenados en COCO.
4. Comenzar el entrenamiento en nuestro conjunto.
5. Validar usando el servidor de Codalab (https://competitions.codalab.org/competitions/2014).

*** Entrenamiento tras 130 épocas: ***

Parámetros: min-input: 288, max-input: 512

AP (Pascal VOC 2007): ~0.68 mAP (COCO 2017): 0.38

*** Finetuning ***

–> 288,512,ig06,xywh2

- 17 épocas (early stopping) congelando todo menos las 10 últimas capas. Lr inicial de 1e-3, batch size de 12 (train_hist_finetuning). Loss: 53.42
- 80 épocas. Todo descongelado. Batch size 8. Lr inicial 1e-4.

mAP: 0.39

–> 416,512,ig07,xywh2

- 25 épocas congelando primeras 74 capas. Lr inicial de 1e-3. Batch_size de 8. Warmup epochs = 3. Loss: 29.8 Logs: finetuning-30 Tiempo estimado por época: 730s

- ?

–>416,672,ig07,xywh2

- 30 épocas congelando todo menos los 3 bloques de detección. Lr inicial 1e-3. Batch_size de 8. Warmup 4 Loss: 36 Logs: tiempo estimado por época: 650s
- 70 épocas todo descongelado, 416,512, bs8,lr1e-4 Loss: 24 Tiempo: evaluación mAP: 0.4053 AP: 0.7155

*** Modelo base ***

- Cargando backend.h5 tal cual y haciendo finetune 10 épocas (para la última capa de cada bloque de detección). Parámetros: input 416, ig0.5, min-max 416,416, obj0.5, nms0.45, jitter 0.0, xywh1, lr 1e-3 mAP@.5:.05:.95: 0.0181 AP@0.5: 0.0818

Apéndice: Funcionamiento del código

Bibliografía

Dataset WIDERFACE (Yang, Luo, Loy, & Tang, 2016).

1 2 3

Yang, S., Luo, P., Loy, C. C., & Tang, X. (2016). WIDER FACE: A Face Detection Benchmark. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.