Detección de caras usando una red YOLOv3 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.

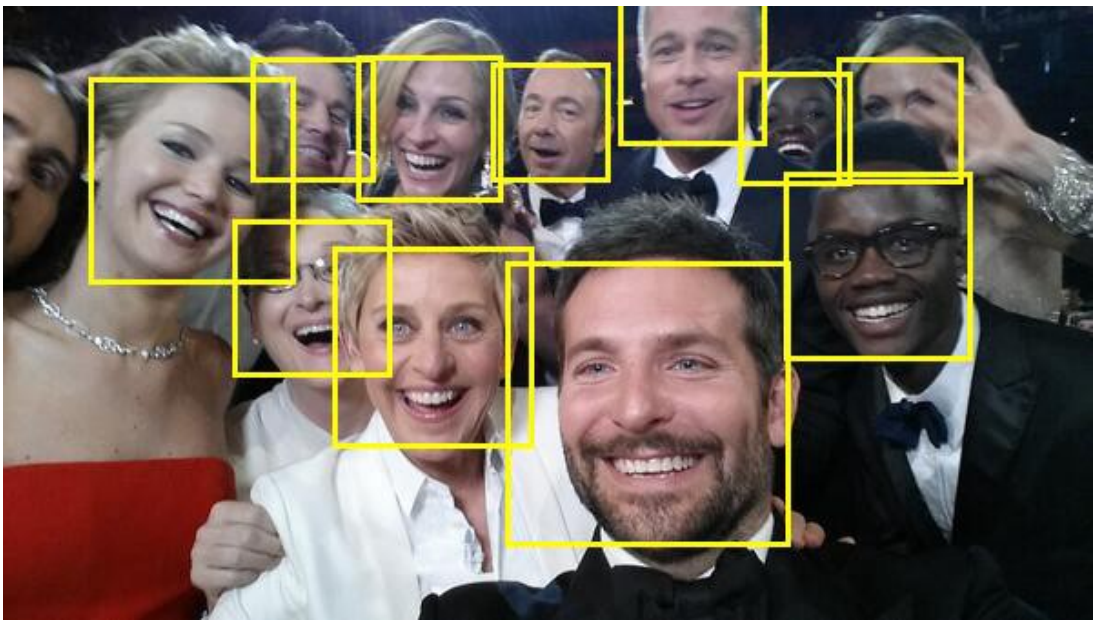


Figura 1: Ejemplo de detección de caras

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.

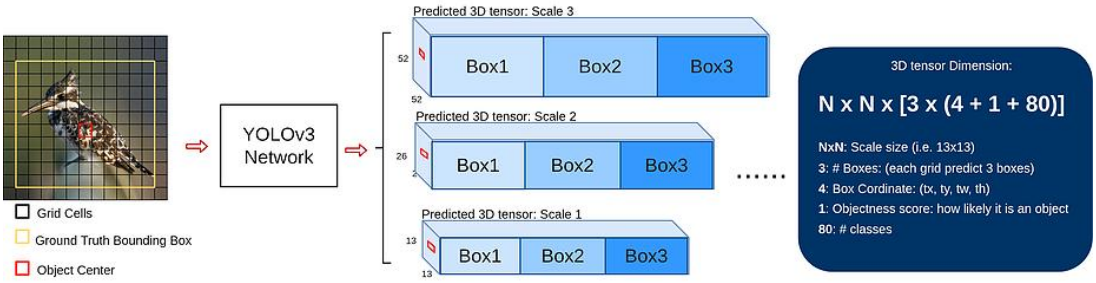


Figura 2: Funcionamiento general

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.

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 red conforme es más profunda más le cuesta detectar objetos pequeños. YOLO resuelve esto usando una estructura de detección piramidal

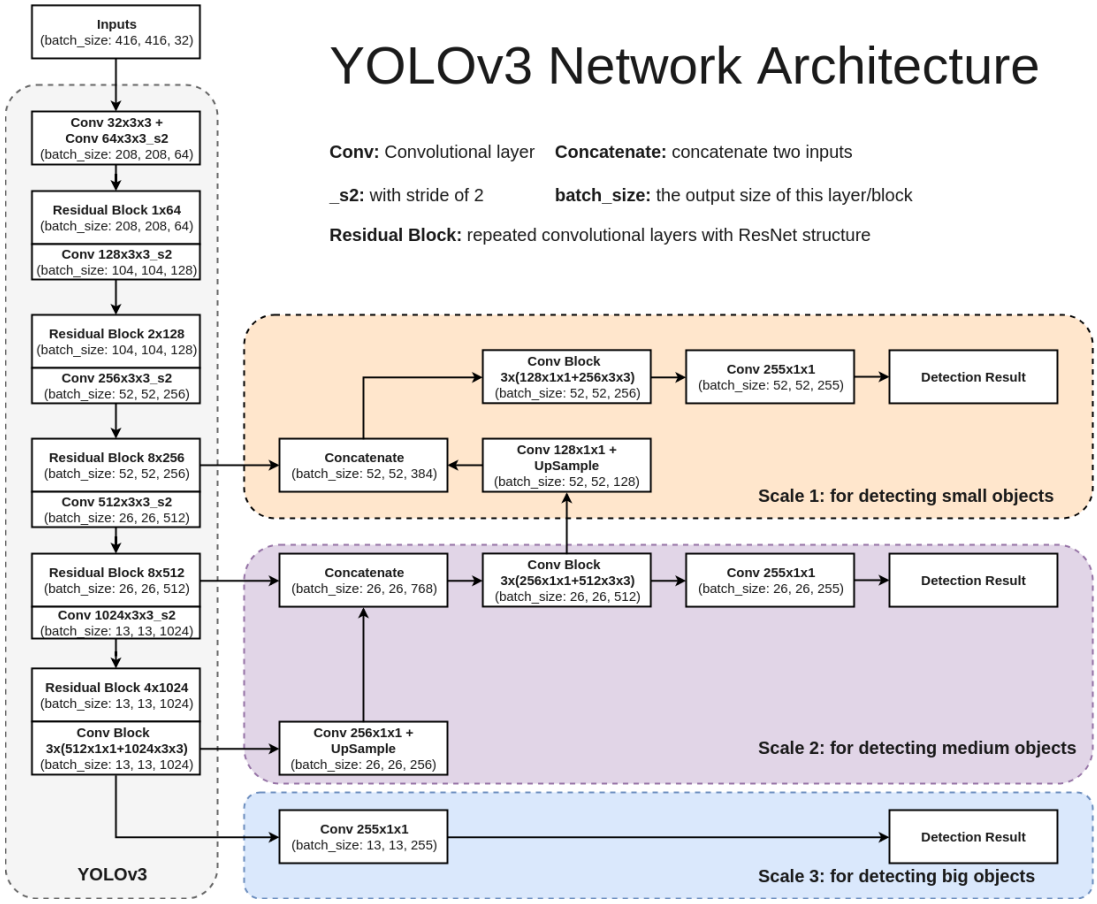


Figura 3: Arquitectura de YOLOv3

	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 4: Darknet-53

(Feature Pyuramid Network) que se encarga de detectar en 3 escalas distintas (pequeño, mediano y grande).

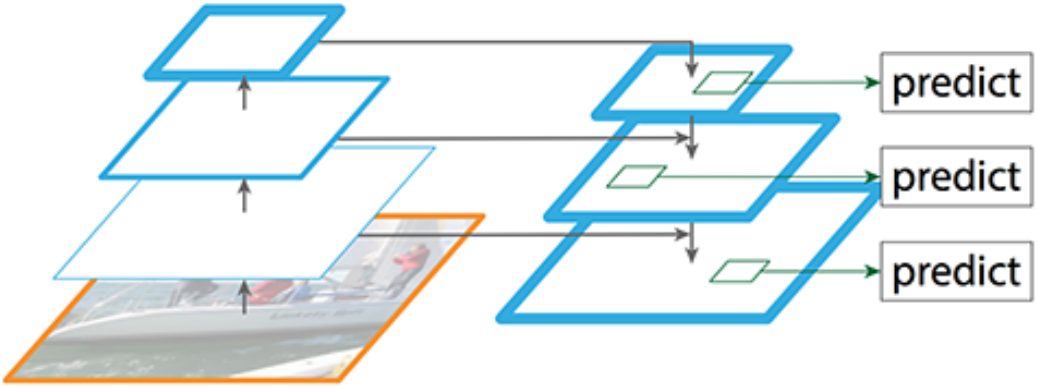
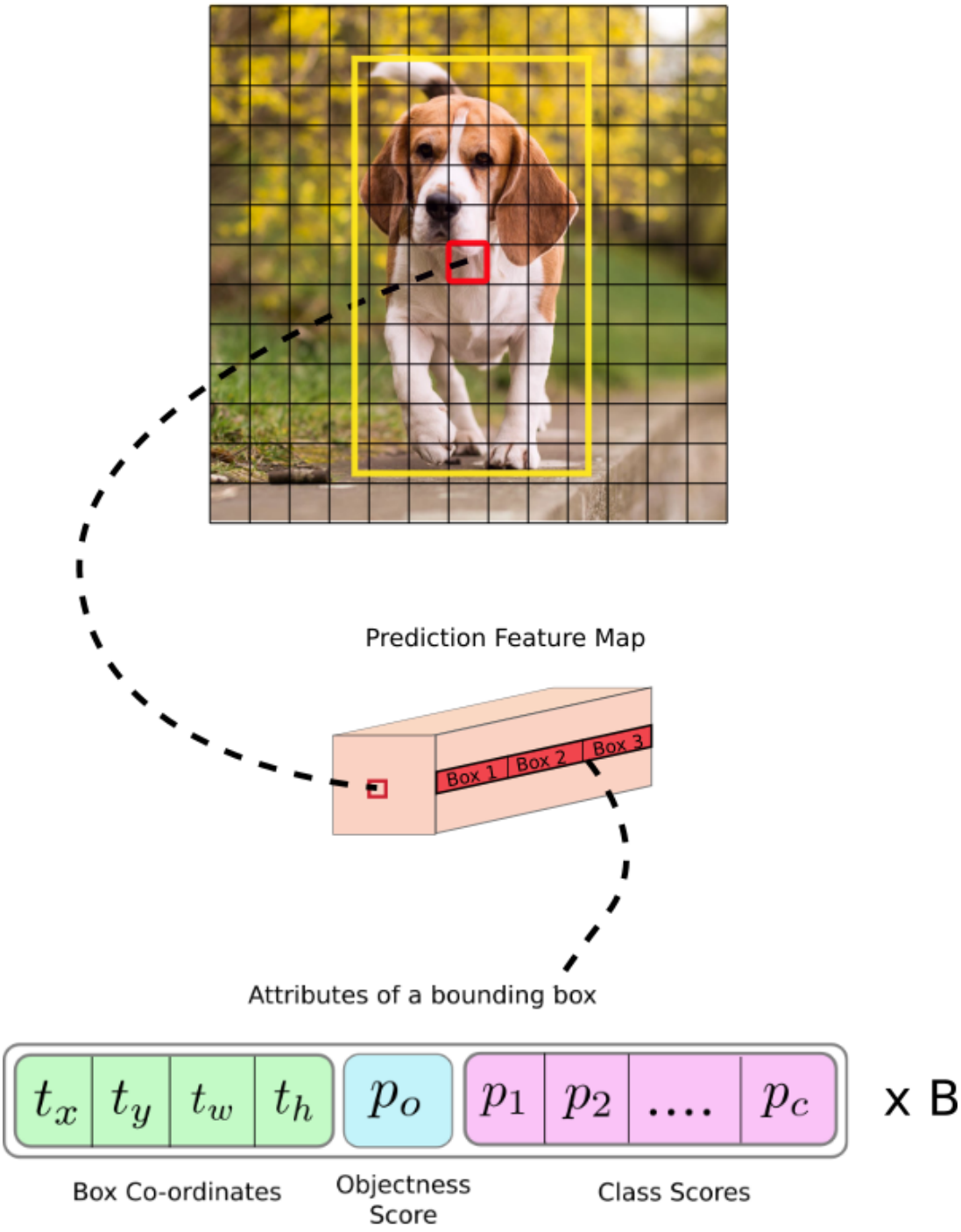


Figura 5: Feature Pyramid Network

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.

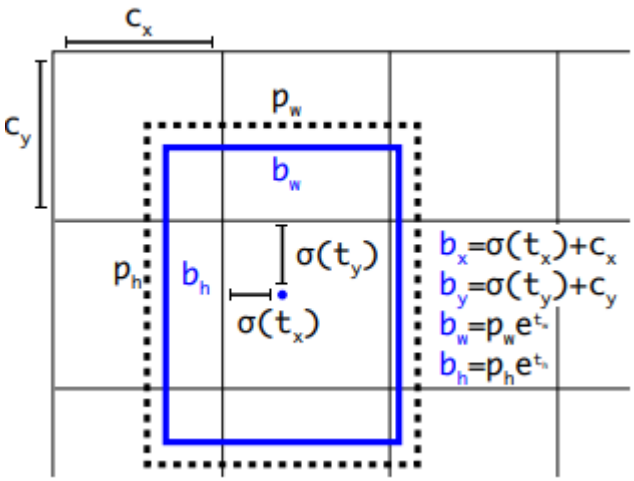


2.4.1. Caja

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.

3. Entrenamiento

3.1. Pesos preentrenados COCO

Hemos obtenido los pesos de YOLOv3 después de ser entrenada en el dataset COCO. Es un dataset con más de 200.000 imágenes con objetivos etiquetados, 91 clases, con 1.5 millones de objetos; en este dataset se encuentra la clase "persona".

Obviamente para la tarea de reconocer caras no es necesaria toda la información de distintos objetos que haya reconocido en COCO, pero como ha aprendido a reconocer personas de un dataset potente podemos aprovechar eso como base para nuestro detector, si bien es cierto que puede que cueste debido a que en

algunas imágenes de personas en COCO no aparecen sus caras o están lejos (están centradas en personas).

3.2. Dataset objetivo WIDERFACE

Vamos a usar el dataset WIDERFACE para entrenar y evaluar la red, el dataset de entrenamiento contiene 32.203 imágenes con 393.703 caras con bounding boxes anotadas que incluye una gran variedad conforme a la forma de las caras: en número, escala, pose, expresión, con maquillaje, distinta iluminación...

Para la evaluación se usa un dataset de 10.000 imágenes de distribución similar al de entrenamiento pero con imágenes nuevas para comprobar el buen funcionamiento de la red.

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-`

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).

5. Cosas

- Cambiar optimizador a SGD, RMSprop,...
- Métrica AP es AUC.

After doing some clustering studies on ground truth labels, it turns out that most bounding boxes

---

6. Consideraciones previas al uso de la red

Como ya hemos comentado, utilizaremos la red YOLOv3 para realizar detección de caras en imágenes. En particular, emplearemos esta implementación en Keras. Para tener un entorno de desarrollo adecuado se necesita hacer los siguiente.

1. En primer lugar, es necesario generar las *anchor boxes* para nuestro dataset. Ya dijimos que la red YOLOv3 predice *offsets* respecto a estos valores predeterminados, por lo que si queremos entrenar la red con imágenes de nuestro nuevo conjunto debemos proporcionar estas cajas prefijadas. Para ello, utilizamos el fichero `gen_anchors.py` que simplemente aplica el algoritmo de k-medias en el conjunto de entrenamiento para predecir el 3 *anchor boxes* en cada escala, dadas en función del alto y del ancho. El resultado es el siguiente:

$$[[2, 4, 4, 8, 7, 14], [12, 23, 20, 36, 35, 56], [56, 95, 101, 149, 177, 234]]$$

2. Para trabajar con las anotaciones de *ground truth* es necesario convertirlas al formato VOC que maneja la implementación proporcionada. Para ello utilizamos el script `utils/convert_annot.py`, adaptado de este código.
3. Por último, descargamos de este enlace los pesos preentrenados de la red en la base de datos COCO. Estos pesos se corresponden a todas las capas convolucionales, sin contar las capas de detección que dependen del *dataset* concreto que utilicemos.
4. Editamos el archivo `config.json` para establecer la ruta de las imágenes y de las anotaciones, y creamos un cuaderno en Google Colab para las ejecuciones. Este cuaderno puede consultarse en el archivo `yolo.ipynb`. Los detalles sobre el código que contiene se pueden consultar en el Apéndice: Funcionamiento del código.

7. Aspectos de entrenamiento de la red

Estudiamos a continuación las consideraciones más relevantes que hemos hecho a la hora de ajustar la red a nuestro conjunto de datos. Hemos decidido entrenar la red, pues al querer detectar una clase con la que no había sido entrenada anteriormente (en COCO no existe la clase "cara") pensamos que necesitaría ser entrenada de forma profunda en el nuevo conjunto.

Enumeramos los principales parámetros y técnicas que contemplamos.

7.1. Data augmentation

La primera mejora que consideramos es realizar aumento de imágenes para obtener una mayor precisión en el conjunto de validación. Mediante la configuración de los parámetros `min_input_size` y `max_input_size` podemos establecer los tamaños mínimo y máximo de las imágenes, que deberán ser siempre múltiplos de 32. Por limitaciones en el entorno utilizado rara vez podremos superar tamaños de 512×512 para entrenar.

A la hora de entrenar, las imágenes se redimensionan automáticamente cada 10 *batches* a algún tamaño comprendido entre el mínimo y el máximo que sea múltiplo de 32.

También aplicamos transformaciones aleatorias de escala y recorte, cuya intensidad se controla mediante el parámetro `jitter` para el generador de imágenes de entrenamiento. Por defecto la fijamos a 03.

7.2. Tamaño del batch

Debido a las limitaciones en cuanto a la memoria disponible, nos vemos obligados a utilizar un *batch size* de 8 para las imágenes de entrenamiento. Congelando un número elevado de capas podemos llegar a un *batch size* de 12.

7.3. Optimizador y learning rate

Empleamos el optimizador Adam para compilar el modelo. Comenzamos a entrenar los modelos con un *learning rate* elevado de 0001, que es el valor por defecto de este optimizador. Disponemos de un *callback* de `ReduceLROnPlateau`, que establece un *learning rate* 10 veces menor cada vez que llevemos dos épocas sin mejorar la función de pérdida. De esta forma conseguimos acelerar la convergencia en las épocas iniciales y ajustar gradualmente los pesos conforme avanzamos en el entrenamiento.

7.4. Épocas de "calentamiento"

El parámetro `warmup_epochs` del archivo de configuración permite especificar un número de épocas iniciales en las cuales las cajas predichas por el modelo deben coincidir en tamaño con los *anchors* especificados. Lo fijamos a 3, y notamos que solo se aplica en las primeras etapas del entrenamiento.

7.5. Umbral de predicción

Internamente la red utiliza el parámetro `ignore_thresh` del archivo de configuración para decidir qué hacer con una predicción. Si el solapamiento entre la caja predicha y el valor de *ground truth* es mayor que el umbral, dicha predicción no contribuye al error. En otro caso, sí contribuye.

Si este umbral es demasiado alto, casi todas las predicciones participarán en el cálculo del error, lo que puede causar *overfitting*. Por el contrario, si este valor es demasiado bajo perderemos demasiadas contribuciones al error y podríamos causar *underfitting*. El valor por defecto es 05.

7.6. Cálculo de la función de error

Mediante los parámetros `obj_scale`, `noobj_scale`, `xywh_scale` y `class_scale` podemos fijar la escala en la que afecta cada parte al error total. El primero se refiere al error dado al predecir que algo es un objeto cuando en realidad no lo era, y el segundo a la acción contraria. El tercero controla el error en la predicción de las cajas frente a los valores reales, y el cuarto el error en la predicción de clase. Los valores por defecto son 5, 1, 1 y 1, respectivamente.

7.7. Otros callbacks

Disponemos de un callback de `ModelCheckpoint` que va guardando un modelo con los mejores pesos obtenidos hasta el momento, de forma que podemos reanudar el entrenamiento por donde nos quedásemos. También tenemos un callback de `EarlyStopping` para detener el entrenamiento si no disminuye el error en 7 épocas.

8. Modelos entrenados y evaluación

Los parámetros utilizados para todas las evaluaciones son `obj_thresh = 0.5` y `nms_thresh = 0.4`. El primer parámetro se refiere al umbral a partir del cual se considera que un objeto detectado es realmente un objeto (el resto se descartan), y el segundo controla el umbral de la supresión de no máximos realizada para eliminar detecciones solapadas.

Mostramos ahora los modelos finales que hemos obtenido. Hemos hecho más pruebas de las que se reflejan aquí, pero la mayoría han sido infructuosas.

8.1. Modelo base

En primer lugar generamos un modelo base con el que compararemos nuestros intentos de mejora. Se trata simplemente de un modelo con los pesos preentrenados de COCO y una capa de detección añadida en cada escala. Lo entrenamos durante 10 épocas, congelando todas las capas excepto las 3 añadidas. Utilizamos los parámetros por defecto y no realizamos *data augmentation*, y elegimos un tamaño de entrada de 416×416.

Al evaluar este modelo obtenemos lo que ya esperábamos: unos resultados mediocres. Esto es normal, ya que la red no estaba entrenada originalmente para reconocer caras. Las métricas de evaluación obtenidas son:

```
mAP@.5:.05:.95: 0.0241
AP@0.5: 0.0818
```

8.2. Modelo 1: entrenamiento completo

La primera prueba que hicimos fue entrenar el modelo completo partiendo de los pesos de COCO, de nuevo utilizando los parámetros por defecto. Esta vez sí empleamos aumento de datos, estableciendo los límites de las dimensiones en 288 y 512. Este modelo fue entrenado durante unas 130 épocas, a razón de unos 700 segundos por época. La pérdida fue disminuyendo hasta estancarse en un valor cercano a 19. Intentamos reiniciar el entrenamiento partiendo de un *learning rate* más elevado para escapar del óptimo local, pero este enfoque no surtió efecto.

La evaluación para un tamaño de entrada de 416×416 fue:

```
#TODO HACER!!!
```

Vemos que mejora bastante al modelo base. Si evaluamos este mismo modelo con un tamaño de entrada de $1024x1024$ obtenemos una precisión bastante mayor. A cambio debemos esperar bastante más tiempo a que se realicen las detecciones en las imágenes.

```
AP@0.5: 0.6656
mAP@.5:.05:.95: 0.3760
```

8.3. Modelo 2: finetuning en los bloques de detección

Intentamos ahora realizar *finetuning* en los bloques de detección de imágenes. Fijamos el valor de `ignore_thresh = 0.7` y aumentamos al doble la contribución al error de las diferencias entre las cajas predichas y las verdaderas, haciendo `xywh_scale = 2`. Hacemos todo esto para intentar mejorar la precisión. Ahora cargamos los pesos de COCO y dividimos el entrenamiento en dos partes:

1. Congelamos toda la red excepto las 4 últimas capas de cada escala. Además, establecemos el *batch_size* a 12 y permitimos que las dimensiones de entrada fluctúen entre 416 y 672 (podemos aumentar el límite superior porque hemos congelado la mayoría de las capas). Entrenamos el modelo durante 30 épocas y nos estancamos en una pérdida alrededor de 36. El tiempo estimado por época es de 650s.
2. Ahora descongelamos todas las capas y entrenamos el modelo durante unas 70 épocas. Volvemos a establecer los límites de entrada en 416 y 512 y el *batch size* a 8, y esta vez partimos de un *learning rate* inicial de 10^{-4}. Obtenemos una pérdida de 24.

El resultado de la evaluación del modelo con tamaño de entrada 1024×1024 es el siguiente:

```
AP@0.5: 0.7255
mAP@.5:.05:.95: 0.4053
```

Vemos que supera al modelo anterior en ambas métricas, por lo que los ajustes realizados han surtido efecto.

8.4. Modelo 3: congelar extractor de características

El último intento exitoso de mejora del modelo es parecido al anterior, pero esta vez congelamos únicamente el extractor de características de la red: las 74 primeras capas.

1. En la primera etapa entrenamos 25 épocas partiendo de un *learning rate* de 0001 y obteniendo una pérdida de 30. El tiempo estimado por época es de 730s.
2. A continuación entrenamos el modelo completo durante 50 épocas, llegando a una pérdida de 25.

Al evaluar con tamaño de entrada 1024×1024 obtenemos los siguientes resultados:

```
AP@0.5: 0.7135
mAP@.5:.05:.95: 0.3945
```

Vemos que se obtiene un resultado muy similar al del modelo anterior, pero un poco por debajo. Sin embargo, este modelo ha sido entrenado durante unas 30 épocas menos.

9. Índice

- problema a resolver
- bases de datos usadas (COCO, WIDER)
- red usada: yolov3
- medidas de precisión (+ competición codalab)
- ejemplos de detección y grabaciones.
- Conclusiones y otras propuestas (otras redes)
- Funcionamiento del código (explicar)

Apéndice: Funcionamiento del código

9.1. Construcción del modelo

9.2. Generadores de imágenes

9.3. Entrenamiento

9.4. Predicción

9.5. Evaluación

Bibliografía

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

Info YOLO 3 YOLOv3 paper Info YOLO 2 COCO dataset WIDERFACE

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)*.