VISIÓN POR COMPUTADOR

DETECCIÓN DE CARAS CON YOLOv3 Y WIDERFACE

PROYECTO FINAL

Víctor Manuel Arroyo Martín
Pedro Gallego López

Doble Grado de Ingeniería Informática y Matemáticas

Universidad de Granada
Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Introducción

El objetivo del proyecto es explorar la capacidad de adaptación de la red YOLOv3 para detectar caras habiendo sido preentrenada con la base de datos COCO (que no contiene esta clase de objeto). Para abordar el proyecto se ha cogido una base de datos de imágenes con caras etiquetadas, la base de datos es **WIDERFACE**.

Como cabe esperar, por el motivo del proyecto, COCO es una base datos que no incluye la categoría de objeto "cara". Sin embargo, es una base de datos muy completa: tiene hasta 91 categorías distintas, entre ellas la categoría de personas, lo que en principio podría ayudar a nuestra tarea de encontrar caras. Debido a estos motivos se ha decidido preentrenarla con esta base de datos.

Para ejecutar el proyecto vaya al Apéndice de ficha técnica situado al final del documento.

COCO (Common Object in Context)

Es una de las bases más populares usada para detección y segmentación. Es una base de datos con imágenes tomadas del día a día con un total de 91 clases, entre ellas la clase "persona"que nos ayudará para la detección de caras con más de 250000 instancias de ésta.

WIDERFACE

Es un dataset creado para la detección de caras con más de 12000 imágenes para el entrenamiento y 3000 para la validación sumando más de 300000 caras con mucha variabilidad: pose, punto de vista, maquillaje etc.
Hay también un conjunto disponible para el test pero sólo liberan el ground truth para las competiciones actualmente inactivas, así que usaremos el conjunto de validación que nos proporciona la web de WIDERFACE para evaluar nuestro modelo.
El dataset viene separado en 62 carpetas cada una con una temática de foto donde aparecen caras: desde fotos de manifestaciones hasta fotos de carreras de coches.
Lo hemos obtenido del repositorio oficial de WIDERFACE.

Para la práctica se ha usado únicamente la mitad del dataset de entrenamientos por temas de tiempo y cómputo. Se han despreciado en el entrenamiento carpetas enteras y esto hará que tengamos un peor rendimiento que si el estudio se hubiera hecho con el conjuunto entero.

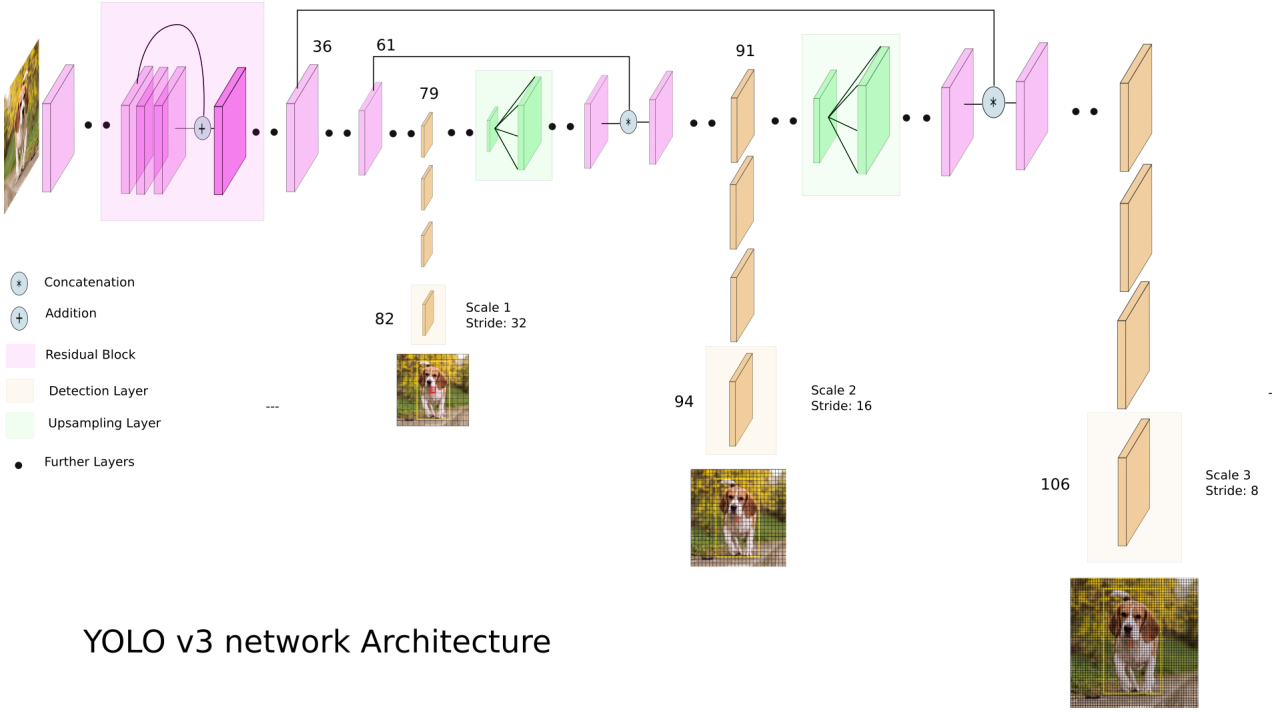
YOLOv3

You Only Look Once v3 tiene una arquitectura completamente convolucional dirigida a la detección de objetos y que puede ejecutarse en tiempo real, siendo uno de los más precisos a la

fecha con estas características. YOLOv3 es una mejora de los anteriores YOLO. Comparando con las anteriores versiones, v3 soporta detección multi-escala, tiene una red extractora de características más potente y cambios de mejora en la función de pérdida. Como resultado, podrá detectar con más precisión más objetos.

Arquitectura de red

Vamos a explicar la arquitectura general de este modelo con un diagrama a gran escala. El sistema puede ser dividido en dos grandes componentes: el extractor de características y el detector, ambos multiescala. Después estas características se pasan a tres o más ramas del detector para obtener las bounding boxes y la información de la clase.



YOLO v3 network Architecture

YOLOv3 tiene 106 capas totalmente convolucionales: 53 de DarkNet y 53 para la detección. Esto se refleja en la imagen superior.

DarkNet53

Es el extractor de características usado por YOLOv3. Éste toma la idea de ResNet de saltarse conexiones para ayudar que las activaciones se propaguen a las capas más profundas sin el gradiente descendiente. Podemos verlo en el siguiente diagrama:

	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			

Table 1. **Darknet-53.**

Para explicarlo, vamos a considerar las capas de cada rectángulos como bloques residuales. La red por completo es una cadena de múltiples bloques con algunas capas convolucionales de stride 2 intercaladas para reducir la dimensión. Dentro del bloque hay una estructura de cuello de botella (convolución 1x1 seguida de una 3x3) y una conexión skip que explicábamos anteriormente. También, después de cada capa se añade una BatchNormalization y activación Leaky ReLU.

Detección en tres escalas

La característica más destacada de v3 es que realiza detección a tres diferentes escalas. YOLO es una red totalmente convolucional y su salida es generada aplicando un kernel 1×1 en un mapa de características. Pues bien, en YOLOv3, la detección se hace aplicando kernels de detección 1×1 en mapas de características de tres distintas dimensiones en tres sitios distintos de la red.
La dimensión de este kernel de detección es $1 \times 1 \times (B \times (5 + c))$ donde B es el número de bounding boxes que una célula puede predecir en el mapa de características y C es el número de clases, en nuestro caso sólo hay 1. En el caso de YOLOv3 entrenado en COCO, $B = 3$ y $C = 80$ dando unas dimensiones de $1 \times 1 \times 255$.

YOLOv3 también hace uso de las llamadas "anchor box" que tienen un ratio de aspecto

predefinido para antes de empezar el entrenamiento. Esto se obtiene con el $K-Medias$ en el dataset completo. La función de pérdida que usa YOLO está compuesta de los siguientes términos:

1. Coordinate loss: generado por una caja que no cubre exactamente a un objeto
2. Objectness loss: debido a una predicción IoU errónea en una caja
3. Classification loss: debido a desviaciones de predecir '1' en la clase correcta y '0' para todas las demás clases del objeto en esa caja
4. Contraction loss: resultante de las primeras etapas al haber diferencias entre la caja predicha y el anchor box

$$\begin{aligned}
&\lambda_{coord} \Sigma_{i=0}^{S^2} \Sigma_{j=0}^{B} 1_{ij}^{obj} \left[\left(x_i - x_i'\right)^2 + \left(y_i - y_i'\right)^2 \right] \\
&+\lambda_{coord} \Sigma_{i=0}^{S^2} \Sigma_{j=0}^{B} 1_{ij}^{obj} \left[\left(\sqrt{w_i} - \sqrt{w_i'}\right)^2 + \left(\sqrt{h_i} - \sqrt{h_i'}\right)^2 \right] \\
&+\Sigma_{i=0}^{S^2} \Sigma_{j=0}^{B} 1_{ij}^{obj} \left(C_i - C_i'\right)^2 \\
&+\Sigma_{i=0}^{S^2} \Sigma_{j=0}^{B} 1_{ij}^{noobj} \left(C_i - C_i'\right)^2 \\
&+\Sigma_{i=0}^{S^2} 1_{i}^{noobj} \Sigma_{céclasses} \left(p_i(c) - p_i'(c)\right)^2
\end{aligned}$$

→ Función de pérdida de YOLO-v1. Para v2 y v3 las predicciones directas en altura y anchura y la raíz cuadrada fueron reemplazadas con una predicción de escala residual para hacer el argumento de pérdida proporcional al error relativo en vez de absoluto.

El IoU (Intersection over Union) mide la calidad de una bounding box con el objeto que pretende medir. Vale 0 si la caja falla por completo el objeto o 1 si se ajusta a la perfección.

Desarrollo del proyecto

En esta sección vamos a hablar del procedimiento seguido para crear los modelos y las mejoras que hemos implementado junto a sus resultados.
Primero veamos las posibilidades que nos ofrece el archivo de configuración json obtenido del github y cuáles hemos usado.

Métricas

Se ha usado la métrica Mean Average Precision (mAP). Para calcular el mAP es necesario calcular los valores de *Precision* y *Recall*. Para calcularlos, necesitamos identificar verdaderos positivos, falsos positivos y falsos negativos. Para obtener los verdaderos positivos, utilizamos IoU. Usando IoU, tenemos que identificar si la detección (un positivo) es correcta (verdadera) o no (falsa). Lo más normal es usar como threshold 0.5, es decir, $IoU > 0{,}5$. Sin embargo, en COCO se recomienda medir a través de varios IoU threshold, de 0.5 a 0.95 por ejemplo.

$$Precision = \frac{verdaderos\ positivos}{verdaderos\ positivos + falsos\ positivos}$$

Para calcular el *Recall*, necesitamos contar los negativos. Toda parte de la imagen donde no se ha predicho un objeto es un negativo, y se intentan buscar los falsos negativos.

$$Recall = \frac{verdaderos\ positivos}{verdaderos\ positivos + verdaderos\ negativos}$$

A partir de los valores de *Precision* y *Recall* se calcula una función $pr(t)$, que define una curva PR sobre el plano con valores positivos. Es de la curva PR de donde nace nuestra métrica mAP, esta se calcula de manera aproximada como el área encerrada por debajo de la curva PR

Aumento de datos

Para esta parte, se nos da la posibilidad mediante el parámetro `jitter` regulamos la intensidad del recorte y escala que se le aplica a las imágenes, donde 0 indica que no se aplique.
También, cada 10 batches las imágenes se redimensionan a un múltiplo de 32 comprendido entre los tamaños máximo y mínimo de entrada que hemos establecido.

Tamaño del batch y optimizador

El batch por razones de limitaciones en el entorno, lo hemos establecido a 12 y en algunos entrenamientos a 8.
El optimizador que empleamos es el Adam y hacemos uso de un callback útil llamado `ReduceLROnPlateau` que va disminuyendo el learning rate por 0.1 cada vez que se lleve sin mejorar dos éopcas de forma que aceleramos la convergencia.

Otros parámetros

El parámetro `warmup_epochs` indica el número de épocas en las que hacemos un calentamiento del modelo (sólo al inicio, sin pesos iniciales que no sean los de YOLO preentrenados con COCO), puesto a 3.

Por otro lado está el parámetro `ignore_tresh` que funciona como umbral para decidir si la predicción contribuye o no al error. Si el solapamiento entre la predicción del bounding box y el valor de ground truth lo supera, no contribuye, en otro caso sí. Lo establecemos a 0.7.

Los parámetros `obj_scale, noobj_scale, xywh_scale y class_scale` se refieren a los cuatro errores descritos en la función de pérdida de YOLOv3.

También hacemos uso de dos callbacks que también hemos visto en las prácticas de la asignatura: `EarlyStopping` para detener el entrenamiento si no se mejora el error (disminuye) en un número de épocas establecido, en nuestro caso 7 y `ModelCheckPoint` que guarda los pesos del mejor modelo hasta el momento para renaudar el entrenamiento en cualquier punto de éste que nos quedemos.

Congelación de capas: FINETUNE

Se ha pensado en congelar parte de la red para probar cómo mejora entrenando. Con congelar una capa nos referimos a establecer que esa capa no sea entrenable durante el entrenamiento, es decir, que mantenga los pesos iniciales sin que el back propagation los modifique. Esto es algo interesante ya que como se ha hablado en la introducción sobre la estructura de YOLOv3, se tiene en las primeras 74 capas un extractor de características, el resto de las capas son capas dedicadas a hacer una detección a diferente escala.

El congelar el extractor de características puede ser una buena idea para centrar el aprendizaje en las últimas capas de detección. Hemos pensado en definir un parámetro `FINETUNE` que indique qué capas queremos congelar, este parámetro te congela todas las capas menos las `FINETUNE` últimas, entendiéndose que en las capas no pertenecientes al extractor de características, se tratarán de forma "paralela", es decir: descongelaremos las últimas capas de cada escala.

Para saber qué capas eran estas hemos hecho uso de la función de Keras `model.summary()` que nos muestra una tabla con las capas. En esta tabla se nos muestra a qué capas están conectadas cada una de las capas. De esta forma hemos podido identificar las últimas capas correspondiente a cada escala.

Nota: el hecho de que se concatenen las capas en algún momento del proceso de elaboración del modelo, es decir, que el modelo no sea secuencial, hace que no sea fácil identificar la posición de las capas a nivel de implementación.

Pesos de la pagina web oficial

Hemos obtenido los primeros pesos que hemos usado para entrenar (archivo backend.h5) de la página web oficial de YOLO en la sección donde están subidos los modelos preentrenados con COCO:
En concreto hemos tomado los pesos del siguiente modelo:

Model	Train	Test	mAP	FLOPS	FPS
YOLOv3-608	COCO trainval	test-dev	57.9	140.69 Bn	20

Los archivos necesarios para la conversión son los yolov3.cfg y convert.py del siguiente repositorio:
Repositorio
Y se convierten con la siguiente orden:
`python convert.py yolov3.cfg yolov3.weights model_data/backend.h5`

Modelos entrenados

Primer modelo

Primero hemos empezado con un modelo sencillo del que partir para ir añadiendo mejoras. Este lo hemos hecho sin aumento de datos y hemos congelado el extractor de características (las 74 primeras capas, poniendo el parámetro FINE_TUNE a 8). Por último lo hemos entrenado durante 6 épocas. Para esta primera ejecución se ha tomado el backend (los pesos de YOLO entrenada con COCO). Con estas características nos ha devuelto:

`AP@0.5: 0.0595`

Era de esperar puesto que COCO no posee la clase *cara* y por lo tanto la red no "ha aprendido" a detectarlas. Esto sumado al hecho de entrenarlo durante únicamente 6 épocas, hace que sea algo lógico obtener un mal resultado.

Segundo modelo

Otra de las pruebas que hemos hecho ha sido con aumento de datos pero esta vez congelando todas las capas excepto las 4 últimas de cada escala de detección (mediante el parámetro FINE_TUNE=4) durante 50 épocas. Esto ha resultado ser infructuoso pues se ha congelado demasiada parte de la red y ha resultado en peores resultados que la primera prueba:

`AP@0.5: 0.0439`

De hecho, gracias al `EarlyStopping`, no se entrenó durante las 50 épocas sin mejorar el error sino que paró en la época 14 debido a que no mejoró en las 7 últimas etapas.
Para la siguiente prueba de nuevo congelamos sólo el extractor de características, aplicamos aumento de datos, y bajamos el tamaño del batch a 4 por problemas de memoria en el entorno de ejecución.

Tercer Modelo

Para asegurarnos de que mejora, hemos entrenado esta vez el modelo congelando sólo el extractor de características, aplicando aumento de datos y entrenando durante 10 épocas, obteniendo el siguiente resultado:

`AP@0.5: 0.1367`

Como era de esperar, esta vez sí mejora ya que hemos entrenado más parte de la red ajustándose así mejor a los datos. Hemos obtenido una pérdida de alrededor de 40, que encaja con un resultado bajo como el que nos da. Se empieza a ver una tendencia ascendente positiva de los resultados.

En rojo las verdaderas y en verde nuestros resultados

Cuarto modelo

Viendo que la anterior prueba ha conseguido mejorar el modelo, entrenamos de nuevo esta vez con 70 épocas, con la esperanza de que continue esta tendencia. Se obtienen los siguientes resultados:

```
AP@0.5: 0.4308
```

Con una pérdida alrededor de 32. Estos resultados son asombrosos comparándolos con los anteriores, y por fin vemos que efectivamente realiza un buen desempeño:

En rojo las verdaderas y en verde nuestros resultados

Con esto pesos hemos decidido dejar el modelo, puesto que consideramos que realiza un muy buen rendimiento y que hemos realizado muchos entrenamientos. Creemos que con esto hemos probado la eficacia del modelo para estas tareas, convirtiéndola en un modelo muy potente.

Análisis de resultados y conclusión

Con el estudio de este problema hemos sacado varias conclusiones además de aprender nuevas cosas acerca tanto de la problemática como de factores consecuentes del mismo.

- En primer lugar hay que destacar que no existe en Keras una implementación de esta red, esto ha implicado que parte de nuestro esfuerzo se haya enfocado en crear la propia estructura del modelo así como herramientas para usarlo. Nos hemos enfocado en buscar una implementación del modelo YOLOv3 que entendiésemos y fuésemos capaces de trabajar con ella. A partir de las ideas que nos han brindado los documentos que hemos encontrado (descritos en la bibliografía) hemos creado la estructura de nuestro proyecto, los cimientos del mismo.

  Nos hemos dado cuenta de lo "cómodo" que hubiera sido utilizar alguna red que sí estuviese implementada en Keras. Sin embargo, el ver e investigar sobre cómo hacer una red así de compleja nos ha llevado a aprender bastante.

- Una vez montado todo el código, tenemos que empezar a entrenar nuestra red. Y aquí nos hemos llevado una non grata sorpresa: el modelo es muy tedioso de entrenar, arrojando muy malos resultados si no se entrena durante muchas horas. El tamaño de la base de datos unido a la complejidad del modelo, hace que los entrenamientos sean muy largos, es por ello que nos ha llevado a hacer puntos de guardado y ejecutar poco a poco. Hemos tenido problemas muy serios con google Colab: no dejaba usar la GPU tras cierto uso, (ralentizando el poco tiempo del que disponíamos relativo a llegar a unos pesos buenos), problemas de memoria y demás. Se nos ha hecho cuesta arriba entrenar el modelo, aún sabiendo que este es apto para dar buenos resultados para este problema disponiendo de tiempo y ponencia hardware superiores.

  En definitiva, es un modelo muy tedioso de entrenar con una base de datos de estas dimensiones.

- El punto anterior nos lleva a pensar en la complejidad del problema en comparación al estudiado en prácticas (extracción de características y clasificación de objetos). Comparando con la práctica en la que tuvimos que "diseñar" la red, añadiendo capas sobre otra ya hecha, nos hemos dado cuenta de que la diferencia de la complejidad de tanto en los modelos como en la complejidad del problema es muy grande. En YOLOv3 abarcamos varias funciones de pérdida que hay que minimizar, y el problema de detección de objetos implica el de clasificación, por lo cual solo puede ser más complejo.

  Con esto llegamos a una reflexión sobre cómo un problema cuanto más complejo, puede implicar mucho mayor esfuerzo: estructural en cuanto al modelo y computacional en cuanto al entrenamiento.

Se ha visto en esta práctica lo costoso que es obtener un buen modelo para este problema, ha conllevado una cantidad de tiempo y cómputo muy alta, mucho más de lo que pensábamos. Al final, se ha conseguido llegar a un modelo que realiza un buen desempeño.

Apéndice: Ficha técnica

En este apéndice se hablará de la estructura de los archivos que conforman el proyecto así como su función y su contenido. Además, podrá seguir los pasos para ejecutar el cuaderno por su cuenta, instrucciones al final del apéndice. Comenzamos viendo la estructura, la práctica se ha realizado sobre un cuaderno de Google Colab, así se tendrá en cuenta la estructura desde un directorio de drive donde el "fichero ejecutable" será el cuaderno Colab `.ipynb`:

- `DIRECTORIO_PROYECTO`
  - `widerface` (Directorio)
    - `train` (Directorio)
    - `train_annot` (Directorio)
    - `valid` (Directorio)
    - `valid_annot` (Directorio)
  - `yolo` (Directorio)
    - `utils` (Directorio)
      - `bbox.py`
      - `convert_annot.py`
      - `image.py`
      - `utils.py`
    - `callbacks.py`
    - `gen_anchors.py`
    - `generator.py`
    - `train.py`
    - `voc.py`
    - `yolo.py`
  - `config.json`
  - `eval` (Directorio
  - `models` (Directorio)
  - `CuadernoDelProyecto.ipybn`

Se ha tomado como principal referencia el repositorio de github:

$$https://github.com/experiencor/keras-yolo3$$

Cogiendo muchas ideas de aquí se ha llevado el enfoque de la estructura de nuestro proyecto. Pasamos a explicar los ficheros, que mencionaremos su función a grandes rasgos ya que el código viene comentado y se puede acceder a los ficheros para tener información más concreta (información de cómo funciona alguna de las funciones definidas, por ejemplo).

Distribución de los directorios

Se puede observar que tenemos cinco directorios principales dentro de nuestro proyecto:

1. `PROYECTO/widerface/` → Se alojan los datos de nuestra base de datos, los directorios que caen de él con el sufijo "_annot" son las etiquetas y bounding boxes asociados a las imágenes.
2. `PROYECTO/yolo/` → Aquí tenemos todas las herramientas necesarias para trabajar nuestra red
3. `PROYECTO/yolo/utils/` → Definen herramientas necesarias para llevar acabo los trabajos de la red y sobre la red.
4. `PROYECTO/yolo/models/` → En esta carpeta se guardan los pesos obtenidos.
5. `PROYECTO/yolo/eval/` → Se guarda el archivo `eval.txt`, archivo en el que se guarda el modelo al terminar el entrenamiento y del que se lee antes de la evaluación o detección.

Además cabe destacar el fichero de configuración `config.json`, cogiendo la idea del repositorio, es un fichero que nos sirve para modelar la configuración del modelo y de los conjuntos de entrenamiento y validación, desde este fichero vamos a poder modelar desde el `path` del directorio que queremos leer para leer un determinado archivo, hasta modelar el batch_size usado para la red. Permite de forma cómoda modelar estos parámetros y es por ello que hemos decidido incorporarlo.

Función de los ficheros

En aquellos donde no se haya comentado el código lo suficiente se hablará de forma un poco más extensa en este apartado del apéndice.

bbox.py

Se encarga de manejar los bounding box, se define la clase `BondBox` además de algunas funciones para trabajar con esta clase

- Clase `BoundBox(self, xmin, ymin, xmax, ymax, c=None, classes=None)` que posee los métodos `get_label(self)` y `get_score(self)`.
- Método `_interval_overlap(intervalo_a, intervalo_b)` nos dice, en caso de que se intersequen los dos intervalos, qué cantidad se interseca.
- Método `bbox_iou(box1, box2)` calcula la IoU (intersección sobre la unión) de dos boxes.

$$IoU = \frac{interseccion}{union}$$

- `draw_boxes` dibuja las cajas de detección.

convert_annot.py

Este archivo se utiliza de forma "ajena" al programa. La implementación usada asume que las anotaciones vienen en formato VOC, cosa que no ocurre en nuestra base de datos WIDER-FACE, por lo que es necesario convertir estos ficheros. Se han hecho algunas modificaciones en este fichero que impedían su correcto uso. Así se puede ejecutar este fichero como sigue para obtener la conversión.

```
./convert_annot.py -ap [El path del fichero de anotaciones] -tp [Directorio destino]
-ip [Directorio de imágenes]
```

image.py

Se realizan operaciones sobre la imagen, necesarias para tratar los bounding boxes.

utils.py

En este fichero tenemos una gran cantidad de herramientas, funciones auxiliares, que nos van a permitir llevar a cabo otras tareas más generales. Entre las funciones de mayor importancia está la predicción de los bounding boxes y la evaluación de dicha predicción.

evaluate.py

Este archivo contiene la función evaluate del github original dividida en dos funciones: predict_boxes y evaluate_pascal. Hay además una función que se llama evaluate_coco que es similar a evaluate_pascal

callbacks

Define dos callbacks para el modelo.

gen_anchors.py

Genera los anchors aplicando el algoritmo de $k - medias$ en el conjunto de entrenamiento para predecir 3 anchor boxes en cada escala, dadas en función del alto y del ancho.

generator.py

Define el BatchGenerator y sus funciones

train.py

En este fichero se encuentran las funciones para crear el conjunto de datos de entrenamiento, para crear los callbacks y para instanciar el modelo.

voc.py

En este fichero analizamos las anotaciones, en caso de que las imágenes con sus anotaciones sean correctas, éstas se alojarán en nuevo archivo caché para no tener que estar analizándolo en cada ejecución del cuaderno. Además, se implemente un análisis para cuando la entrada es una única foto en la que queremos detectar.

yolo.py

Se define el modelo YOLOv3 siguiendo la arquitectura descrita anteriormente. La red se construye atendiendo a las 106 capas usando `_conv_block()`, que define un bloque convolucional que puede conllevar una capa de BatchNormalizaion o activación. En este archivo también se trata el apartado de hacer que una capa sea entrenable o no (dentro de `_conv_block()`)

Cuaderno.ipybn

En este cuaderno se realizará la ejecución guiada del proyecto. Este se compone de varias celdas de código donde se definen las funciones y se ejecutan el entrenamiento, evaluación y detección. Para ejecutar la evaluación correspondiente al último modelo entrenado ha de ejecutar cada celda consecutivamente sin saltarse ninguna. En el entrenamiento se guarda el modelo en el drive y en la evaluación lo cargamos de ahí.

Ejecución del cuaderno del proyecto

Para ejecutar el cuaderno necesita crear un acceso directo de la carpeta carpeta proyecto en su Drive dentro de "Mi Unidad". Una vez hecho esto podrá acceder al cuaderno y ejecutar las celdas.

Nota: La cuenta propietaria de los archivo es una cuenta nueva que se ha hecho expresamente para el proyecto. Se alojan en el drive de esta cuenta los archivos del proyecto.
IMPORTANTE: a la hora de compartir la carpeta en los drives de nuestras cuentas personales, tuvimos al principio algunos problemas que se solucionaron aparentemente solos, en donde la ejecución del programa a veces no encontraba algún fichero, en concreto, imágenes de la base de datos.

4.1. Bibliografía

- Paper YOLOv3: An Incremental Improvement
- github.com/experiencor/keras-yolo3
- Dive Really Deep into YOLO v3 article
- YOLO v3 Explained