DETECCIÓN DE CÉLULAS INFECTADAS DE MALARIA

IMPLEMENTACIÓN DE YOLO

Adrián Errea

INTELIGENCIA COMPUTACIONAL

JUNIO 2020

Contenido

DESCRIPCIÓN DEL PROBLEMA

La malaria es una enfermedad causada por parásitos llamados *Plasmodium* y que afecta a 200 millones de personas y causa 400000 muertes cada año en el mundo. Los tipos de malaria que más afectan a los humanos son *Plasmodium falciparum* y *Plasmodium vivax*.

Al igual que con otras infecciones microbianas, la inspección manual de células a partir de análisis de sangre es todavía el método utilizado para la detección de parásitos y su etapa de evolución. Esta metodología ofrece baja reacción, gran coste y poca flexibilidad.

A pesar de que la inspección manual ofrece un rendimiento bajo y susceptible al sesgo humano, el software automatizado permanece inutilizado debido a la gran cantidad de variaciones en las imágenes al microscopio. Sin embargo, una solución automática robusta de clasificación y conteo proveería de enormes beneficios debido a la velocidad y precisión de resultados en comparación con la variabilidad humana.

De esta manera, investigadores y médicos profesionales podrían caracterizar de una mejor forma la etapa exacta para la administración de fármacos al paciente, así como la cuantificación de las reacciones de los pacientes a dichos fármacos.

Intentos previos de automatizar el proceso de identificación y cuantificación de la malaria no han recibido mucha atención debido a la dificultad de replicar, comparar y extender el problema. Además, los autores y centros de investigación raramente publican sus imágenes, lo que imposibilita la replicación de resultados y la evaluación de potenciales mejoras. La falta de un conjunto estándar de imágenes y de un conjunto de métricas utilizadas para clarificar resultados han obstaculizado los avances hasta día de hoy.

Para intentar resolver el problema, se ha puesto a disposición un conjunto de 1364 imágenes (unas 80000 células) de diferentes centros de investigación. Los datos consisten en dos clases de células infectadas (glóbulos rojos y leucocitos) y cuatro clases de células infectadas (gametocitos, aros, trofozoíto y esquizontes). Los investigadores pudieron marcar algunas células como “difíciles” si no estaba claro a qué clase pertenecían.

El objetivo de este trabajo es llevar a cabo la construcción de un modelo de Deep Learning capaz de detectar células infectadas por malaria a partir de imágenes de microscopio obtenidas a partir de análisis de sangre.

YOLO

Vista la descripción del problema, podemos afirmar que se trata de un problema de detección de objetos en imagen. Además, se nos proporcionan dos archivos con información sobre las clases y las bounding boxes correspondientes, lo que nos define claramente el problema.

Debido a esto, se ha optado por implementar el algoritmo ***YOLO***. YOLO es un algoritmo de detección de objetos en tiempo real. Es uno de los algoritmos más efectivos y utilizados para la detección de objetos en imagen. A continuación, vamos a describir su funcionamiento. En concreto explicaremos el funcionamiento de YOLOv3 (se detallará más adelante).

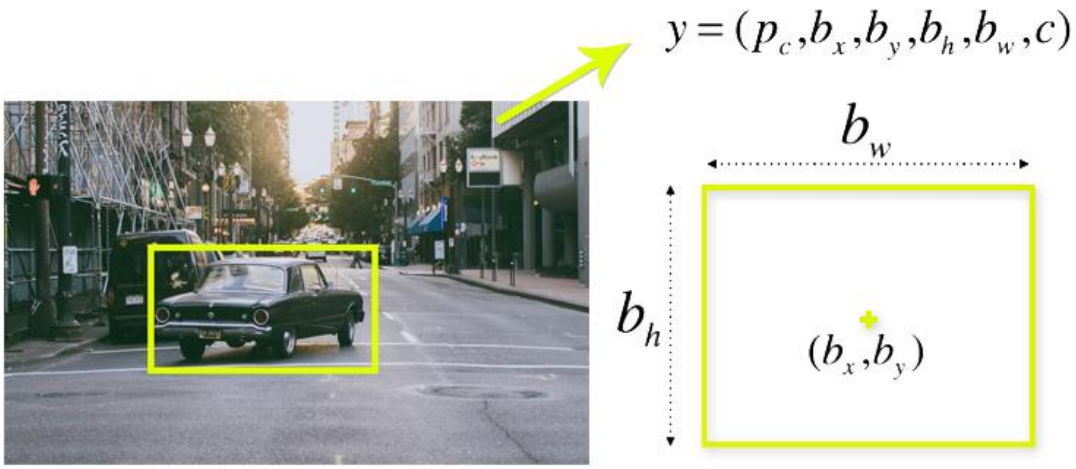
YOLO fue propuesto en 2015 en un estudio de Joseph Redmon et al. (Redmon et al. 2015). Sus siglas vienen del término ***"You Only Look Once"*** y hace referencia a que solo es necesaria una propagación hacia delante para realizar las predicciones de la red neuronal. Para entender bien su funcionamiento, es necesario establecer primero que se va a detectar. En una imagen con múltiples objetos, el objetivo último es detectar la clase de cada objeto y la caja que lo delimita (lo llamaremos como se hace en la literatura inglesa, *bounding box*). Cada bounding box puede ser descrita de varias maneras. Una de las más utilizadas es la siguiente:

$$(b_x, b_y, b_w, b_h)$$

donde:

- b_x indica el centro de la bounding box en el eje X
- b_y indica el centro de la bounding box en el eje Y
- b_w indica la anchura de la bounding box
- b_h indica la altura de la bounding box.

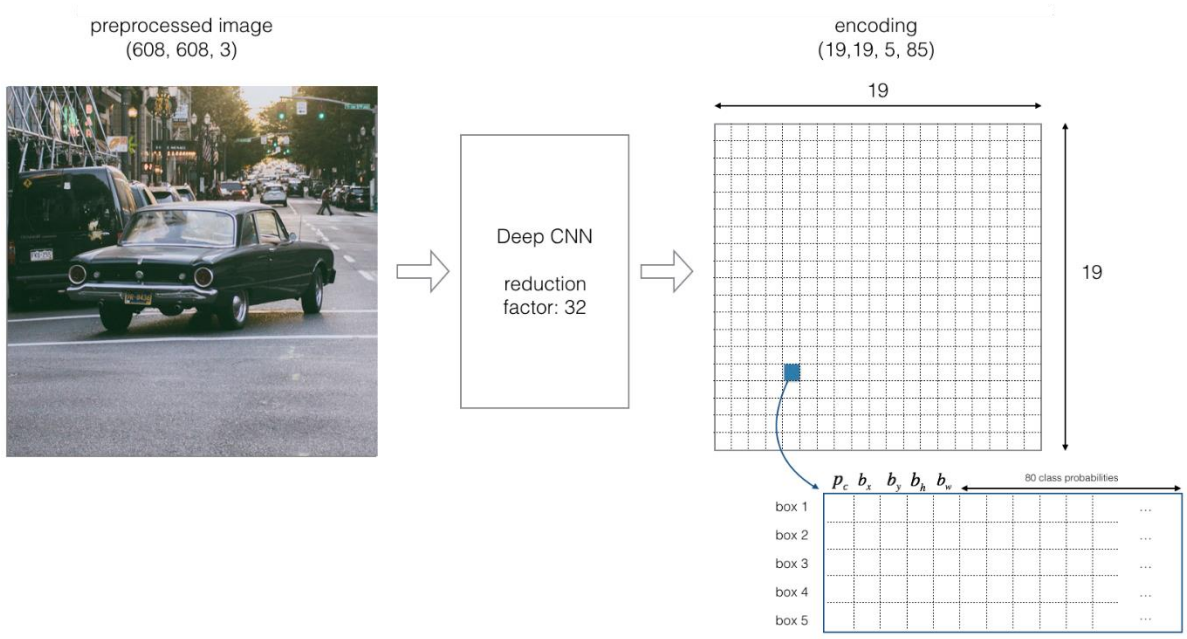
Con esos cuatro valores somos capaces de identificar cada bounding box en una imagen de manera única. Además, como hemos comentado, es necesario asignarle una clase a esa bounding box (c) que tendrá una probabilidad determinada de aparecer en ese propia bounding box (p_c) . Con todo ello podemos ver un ejemplo gráfico en la siguiente imagen con la predicción "*y*" para dicho objeto.



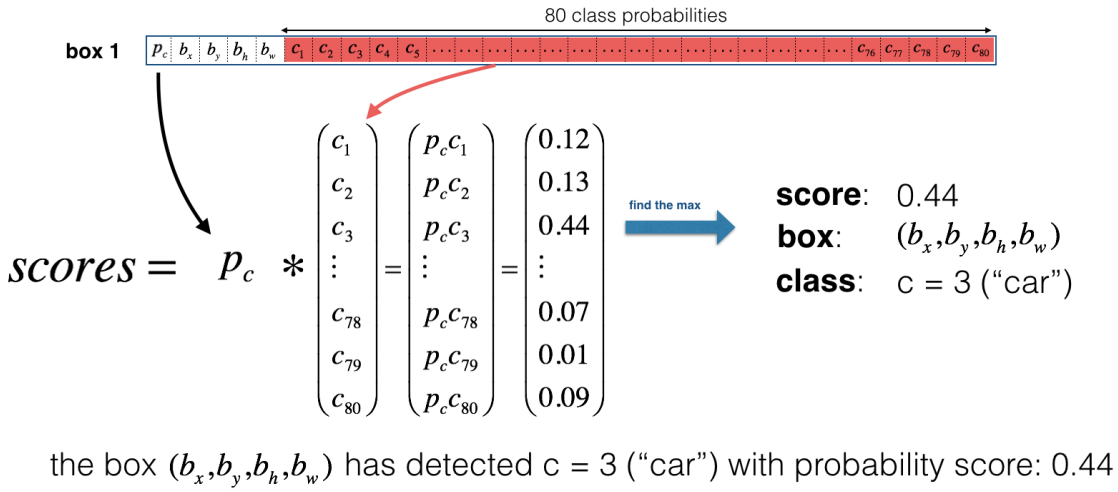
Una vez visto lo anterior, procederemos a explicar cómo funciona el algoritmo YOLO. En primer lugar, se divide la imagen en celdas a partir de una malla (puede ser 3x3, 11x11, 19x19

(la más común), etc.). Cada una de las celdas predecirá un conjunto de bounding boxes predefinidas llamadas ***anchor boxes***. Un *anchor box* es una bounding box con una altura y anchura prefijadas que se utilizan para contener los objetos que queremos detectar. Este término es importante cuando expliquemos el proceso de IoU más adelante.

Suponiendo que hemos predefinido 5 *anchor boxes* diferentes, una representación gráfica quedaría de la siguiente forma.



En este caso la imagen de entrada es de 608x608x3 y tras aplicar una CNN, llegamos a la parte de aplicar la malla (en este caso 19x19) a la imagen. En este ejemplo, tenemos 80 clases para clasificar y cada celda deberá predecir cinco bounding boxes con los valores que hemos visto antes $y = (p_c, b_x, b_y, b_w, b_h, c)$



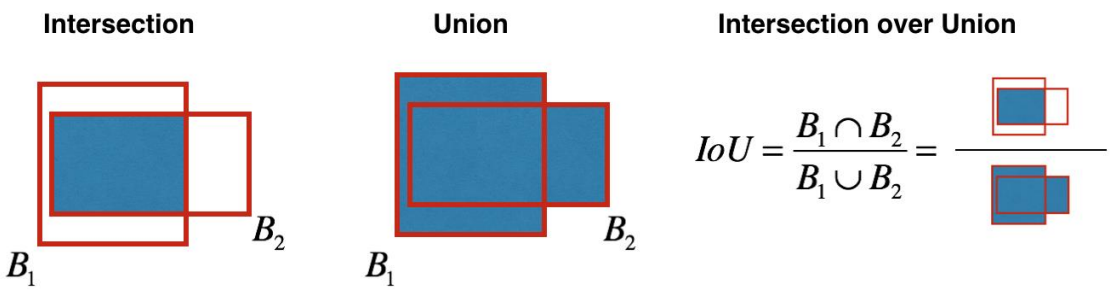
Podemos visualizar lo que está prediciendo el algoritmo dibujando los bounding boxes para cada una de las celdas. En este ejemplo, como cada celda realiza 5 predicciones de bounding box y disponemos de 19x19=361 celdas, por una pasada por la imagen obtendríamos 1805

bounding boxes. Si nos quedamos con aquellas bounding boxes con mayor probabilidad para cada clase obtenemos lo siguiente.

Aunque nos hayamos quedado con las que mayor probabilidad tienen para cada clase a predecir, aún hay demasiadas bounding boxes. Es aquí donde entra la segunda parte del algoritmo: Supresión No Máxima (**Non-Max Supression** en inglés).

Esta función actúa de filtro eliminando las bounding boxes que están solapadas. Su nombre viene de la forma en la que se queda con las bounding boxes con mayor probabilidad de contener un objeto y descarta el resto que estén solapadas con ella.

Aquí es donde vuelve a entrar el término **IoU**: Intersección sobre Unión **(Intesection over Union**, en inglés). Este término consiste en calcular la unión y la intersección de dos bounding boxes y dividir sus tamaños con el fin de obtener una métrica de cuanto de solapadas están dos bounding boxes. Podemos ver un esquema en la siguiente imagen.

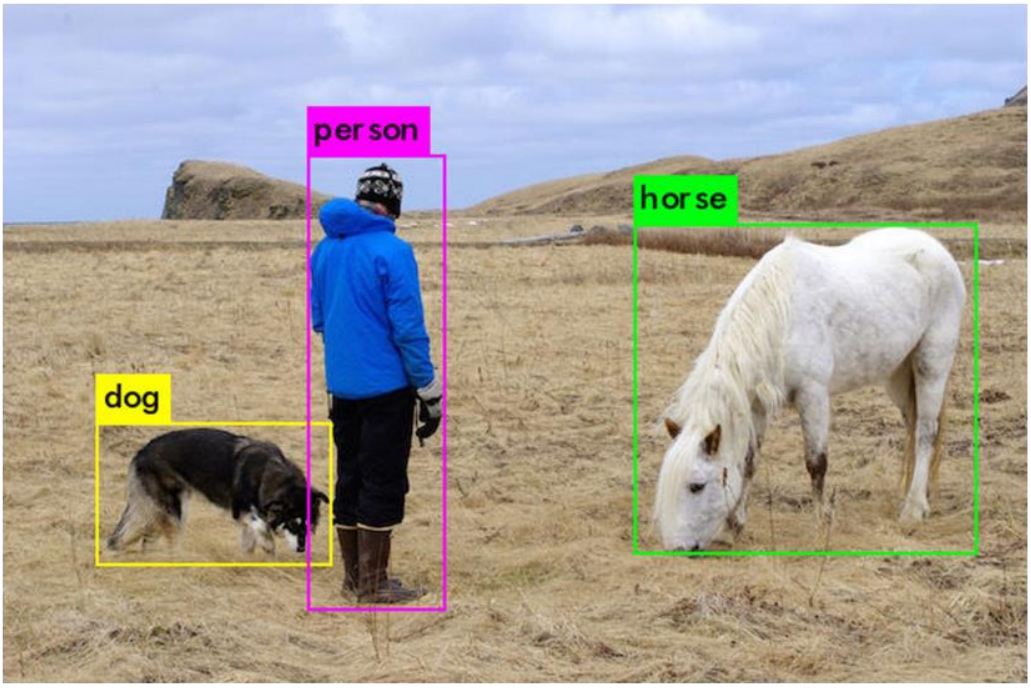


En caso de que el bounding box que estemos tratando sea muy similar al bounding box perfecto que enmarcaría dicho objeto, su IoU sería cercano a 1.

El algoritmo Non-Max Supression, en primer lugar, descarta todos los bounding boxes para cada clase con $p_c < threshold$. Este *threshold* normalmente suele tener valores de 0.5 ~ 0.6. Con ello descartamos de antemano, aquellas bounding boxes que no contienen ningún objeto de interés.

Una vez descartadas dichas bounding boxes, se selecciona aquella bounding box con mayor probabilidad p_c. Esta será la salida predicha para la clase que contiene. Tras ello, se descartan todos los bounding boxes con IoU>=0.5 con dicha bounding box elegida. De esta forma, eliminamos todas aquellas bounding boxes que predecían el mismo objeto que la bounding box elegida, pero la encuadraban de una peor forma. Ese proceso se repite hasta que no queden bounding boxes. El algoritmo Non-Max Supression hay que aplicarlo sobre cada una de las clases que se quieren predecir en una imagen.

Una vez aplicado el proceso anterior obtendríamos un resultado de un bounding box por objeto como se puede observar en la siguiente imagen.



YOLO posee dos grandes ventajas respecto a otros algoritmos de detección de objetos en imagen:

- Debido a que realiza una sola pasada por la imagen cada vez que realiza una predicción, es realmente rápido y por tanto es utilizado en aplicaciones de tiempo real.
- Aprende una generalización de los objetos por lo que obtiene una precisión bastante alta.

A día de hoy se han realizado múltiples mejoras sobre el propio algoritmo y han ido surgiendo diferentes versiones del mismo:

- **YOLOv1:** Es la versión original de YOLO. Utiliza el framework *Darknet* (se explicará más adelante) y está entrenado con el dataset ImageNet-1000. Posee diferentes limitaciones como no ser capaz de encontrar objetos pequeños cuando estos se encuentran muy juntos. Además, si las imágenes a predecir son de otra dimensión a las entrenadas, el algoritmo no generaliza muy bien.

- **YOLOv2:** Esta versión surgió en el 2016 y posee múltiples mejoras respecto a la versión anterior. Entre ellas destacan:
    - *Batch normalization*: Normaliza las salidas de las funciones de activación de las capas ocultas lo que permite reducir el sobreentrenamiento y consigue un efecto de regularización.
    - *Mejora en la resolución:* Las dimensiones de entrada aumentan de 224x224 a 448x448 a la hora de entrenar con el dataset ImageNet, lo que se traduce en una mejora del rendimiento.
    - *Inclusión de Anchor Boxes:* Uno de los cambios más grandes con respecto a la versión anterior.
    - *Características refinadas:* Viene a corregir el problema de YOLOv1 donde no se detectaban objetos pequeños muy juntos. Para ello se utiliza una malla de 13x13 más grande que en la anterior versión.
    - *Entrenamiento multi-escalar*: Debido a que YOLOv1 fue entrenado con imágenes de un solo tamaño y para resolver el problema derivado de la mala generalización, esta versión incluye el entrenamiento con imágenes de diferentes tamaños desde 320x320 hasta 608x608.
    - *Darknet-19:* Esta versión utiliza la arquitectura Darknet-19 compuesta de 19 capas convolucionales, 2 capas Max Pooling y una capa softmax como capa de salida.
- **YOLOv3:** Una de las mejoras más sustanciales del algoritmo.
    - *Predicciones para las bounding boxes:* YOLOv3 da un resultado discreto para cada una de las bounding boxes.
    - *Predicciones por clase:* Esta versión utiliza una función logística en vez de una softmax para la predicción de cada clase. Esto permite la clasificación multiclase.
    - *Redes Características Piramidales (Feature Pyramid Network (FPN), en inglés):* Incorpora el uso de pirámides para la detección de las características principales
    - *Darknet-53:* En este caso utiliza 53 capas convolucionales.
- **YOLOv4:** La última versión publicada y la menos probada y por tanto menos estable.

Visto lo anterior, es importante remarcar que este algoritmo tiene también algunas desventajas:

- No consigue una alta precisión con objetos suficientemente pequeños. Esto es debido a como está implementado el propio algoritmo. Si en cada celda de la malla hay varios objetos, pero solo disponemos de unos pocos *anchor boxes* (teniendo en cuenta que siempre consideramos utilizar YOLOv3), el algoritmo no será capaz de detectar todos los objetos de dicha celda.
- Si dos objetos están muy solapados y muy juntos, tampoco será capaz de detectar más de un objeto.

PREPROCESAMIENTO

Una vez vista la parte teórica, vamos a explicar cómo se han tratado los datos y el preprocesamiento requerido.

Los datos que se nos proporcionan son los siguientes:

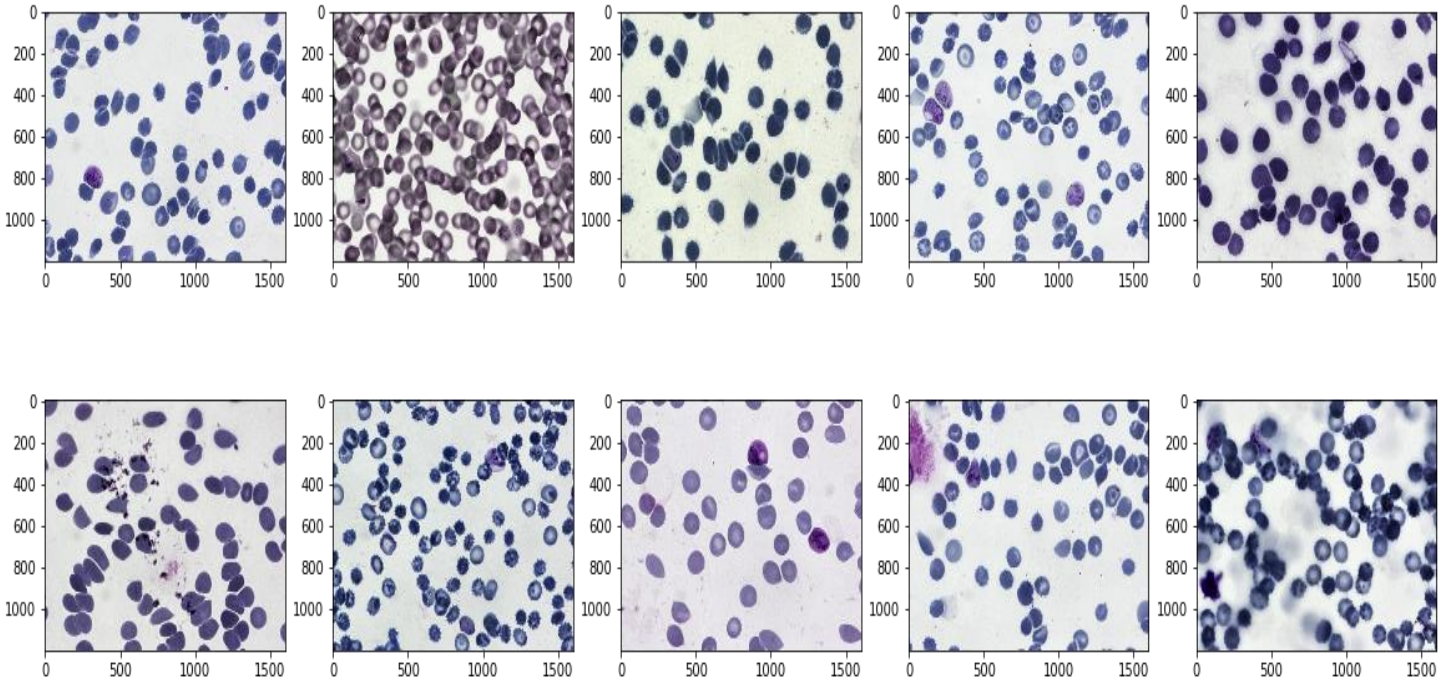
- ✓ 1328 imágenes (~80000 células)
- ✓ Fichero *training.json*
- ✓ Fichero *test.json*

Los datos poseen un desbalanceo claro hacia las clases no infectadas contra las infectadas hasta un ratio de 95/5. Las clases etiquetas son las siguientes:

1. **Glóbulo rojo**
2. **Trofozoíto**
3. **Aro**
4. **Esquizontes**
5. **Gametocitos**
6. **Leucocitos**
7. **"Difícil":** Cuando los investigadores no tenían claro su pertenencia, utilizaban esta clase

Las imágenes están en formato *png* y poseen dimensiones de 1600x1200 y de 1944x1383.

Podemos ver unos ejemplos de las imágenes a continuación:



En los ficheros *json* se guarda por cada imagen:

- ❖ Nombre de la imagen
- ❖ Dimensiones de la imagen

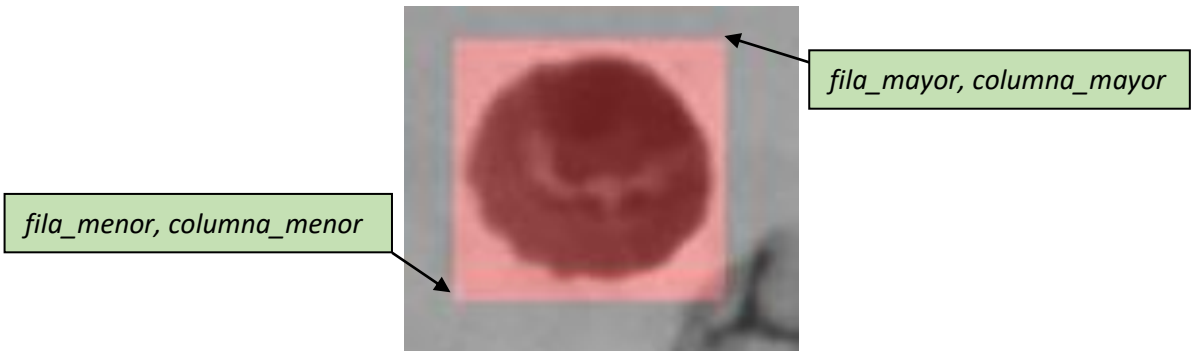
- Bounding boxes contenidas en esa imagen de la forma:

(fila_menor, columna_menor, fila_mayor, columna_mayor, clase)

donde:

- *fila_menor* es la posición en el eje Y de la esquina inferior de la bounding box en la imagen.
- *columna_menor* es la posición en el eje X de la esquina inferior de la bounding box en la imagen.
- *fila_mayor* es la posición en el eje Y de la esquina superior de la bounding box en la imagen.
- *columna_mayor* es la posición en el eje X de la esquina superior de la bounding box en la imagen.

Podemos observar un ejemplo en la siguiente imagen:



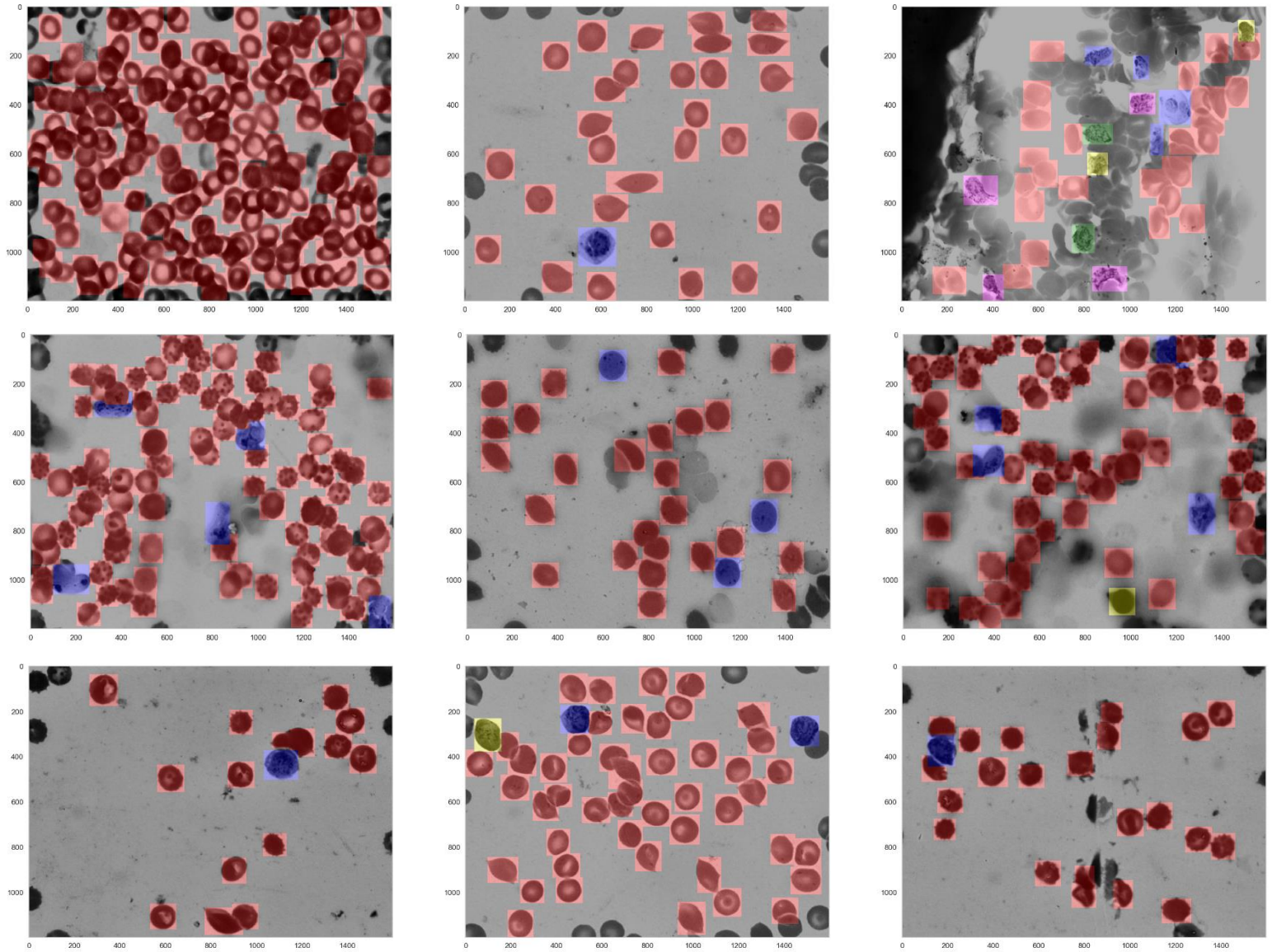
Y su correspondiente valor sería:

{"bounding_box":{"minimum":{"r":1101,"c":507},"maximum":{"r":1248,"c":631}},"category":"red blood cell"}

Lo primero que se realizó fue separar las imágenes en dos conjuntos tal y como nos venían de los ficheros *json*: *train* y *test*. El conjunto de train será dividido más adelante tanto en train como en validación. Tras el primer mapeo de los ficheros json con las imágenes, obtuvimos la siguiente distribución:

- ✓ 1208 imágenes para train
- ✓ 120 imágenes para test

Podemos observar algunas imágenes con sus bounding boxes correspondientes en las siguientes imágenes:



Para poder entrenar nuestro propio dataset con el algoritmo YOLO es necesario, en primer lugar, cambiar el formato de los datos. Esto es debido a cómo interpreta el propio YOLO los datos y como los tenemos almacenados en este momento.

Para cada imagen es necesario tener un fichero en el que en cada línea aparezca la información relativa a cada bounding box de la imagen con el siguiente formato:

(clase, posición_central_eje_X, posición_central_eje_Y, anchura, altura).

donde:

- *clase* es la clase del objeto de la bounding box entre 0 y *numero_clases.*
- *posición_central_eje_X* es la posición en el eje X del punto central de la bounding box en la imagen entre 0 y 1 con respecto a la anchura de toda la imagen.
- *posición_central_eje_Y* es la posición en el eje Y del punto central de la bounding box en la imagen entre 0 y 1 con respecto a la altura de toda la imagen.
- *anchura* es la anchura de la bounding box en la imagen entre 0 y 1 con respecto a la anchura de toda la imagen.

- *altura* es la altura de la bounding box en la imagen entre 0 y 1 con respecto a la altura de toda la imagen.

Un ejemplo de bounding box con este formato seria:

(0 0.7245 0.7411 0.0725 0.0925)

Tras esta transformación tendríamos un fichero por cada imagen que vayamos a entrenar en donde en cada fichero está contenida la información de todos los bounding boxes que contiene.

ENTRENAMIENTO

Una vez que tenemos los datos preparados para entrenar, nuestro objetivo es el de crear un modelo capaz de predecir las células infectadas y no infectadas y localizarlas dentro de una imagen obtenida al microscopio. Para ello, se ha optado por implementar una versión adaptada de YOLOv3 para este problema.

Para hacer este proceso más sencillo, se ha optado por usar ***Darknet***. *Darknet* es un framework de redes neuronales de código abierto escrito en C y CUDA. Soporta computación tanto en CPU como en GPU y fue originalmente implementado por el primer autor de YOLO.

Para poder llevar a cabo el entrenamiento de una manera más rápida se ha optado por utilizar Google Collab. Además, ha sido necesaria la instalación cudNN en la propia sesión de Collab para llevar a cabo el uso de GPU. Una vez instalado y configurado tanto Darknet como Collab ya podemos lanzar nuestro entrenamiento.

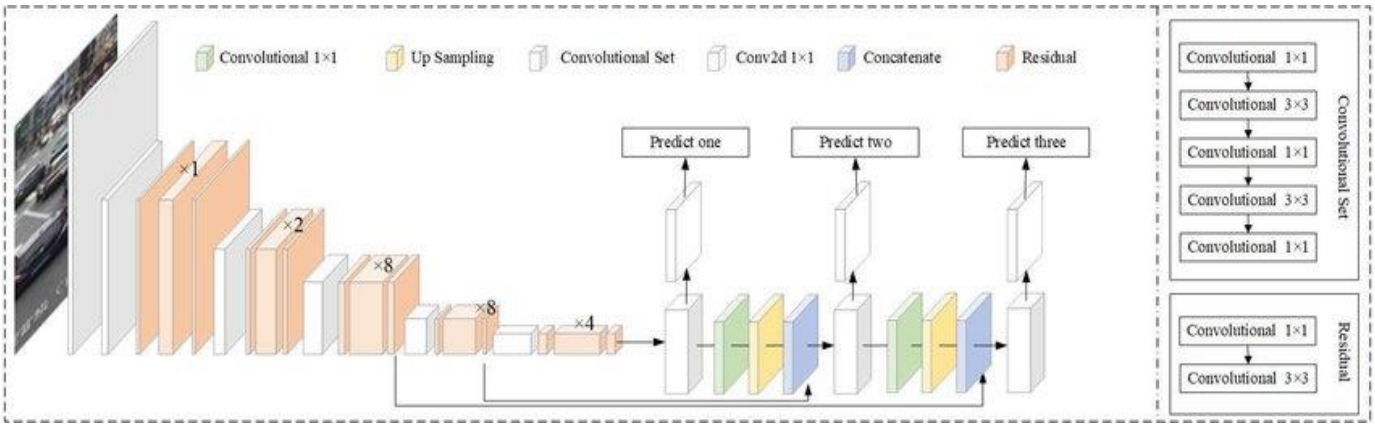
Para ello, es necesario especificar ciertos archivos y configurarlos correctamente para poder ejecutar todo de forma correcta. Los archivos a configurar son:

- *obj.data:* Fichero de configuración que contiene el número de clases y las rutas a los fichero *train.txt*, *test.txt* y *obj.names*.
- *obj.names:* Contiene los nombres de las clases.
- *train.txt:* Contiene las rutas de las imágenes que se usarán para train
- *test.txt:* Contiene las rutas de las imágenes que se usarán para validación
- *yolov3_malaria.cfg:* Contiene la configuración del algoritmo YOLO. Aquí se configura el tamaño del batch, dimensiones de las imágenes, tasa de aprendizaje, capas convolucionales, filtros, anchor boxes, etc.
- *yolov3.weights:* Para empezar nuestro entrenamiento desde una buena posición se cargan los pesos finales con los que está entrenado el modelo YOLOv3 original.

Para la ejecución de nuestro algoritmo se ha utilizado la siguiente configuración:

- Batch=64 imágenes
- Número imágenes train: 1007 imágenes
- Numero imágenes validación: 201 imágenes
- Dimensiones imágenes=608x608
- Tasa aprendizaje=0.001

- *Darknet-53*: 53 capas convolucionales junto con capas residuales, función de activación en las capas convolucionales Leaky ReLu, algoritmo de optimización Gradiente con Momento, así como Batch Normalization.



RESULTADOS

Tras varias iteraciones de entrenamiento se ha conseguido lo siguiente:

```
6524: 3.254687, 4.462 avg loss, 0.001000 rate, 53.254624 seconds, 546540
images
Loaded: 0.000073 seconds

6525: 3.965476, 4.463 avg loss, 0.001000 rate, 53.169279 seconds, 546604
images
Loaded: 0.000050 seconds
```

Del propio algoritmo YOLOv3 podemos interpretar los resultados que nos va ofreciendo tras cada batch de la siguiente manera:

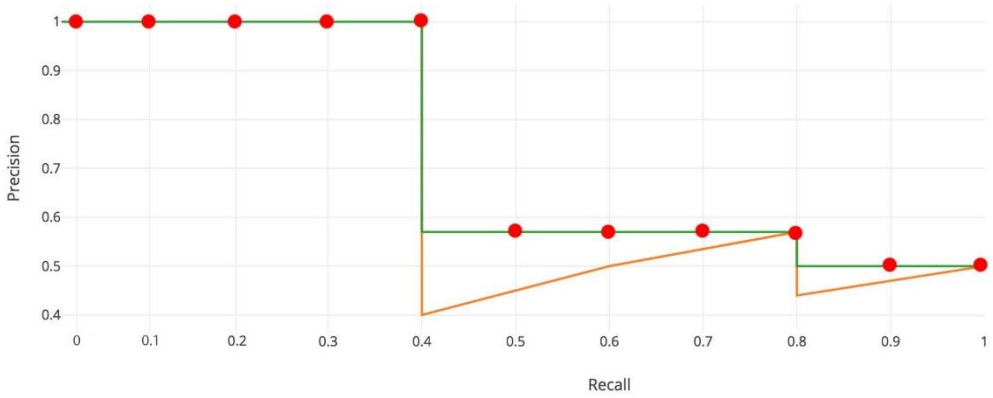
Num_batch: Loss global, Loss medio, actual tasa de aprendizaje, tiempo empleado en ese batch, número total de imágenes entrenadas hasta ahora

En nuestro caso hemos detenido el entrenamiento alrededor de las 6000 iteraciones ya que vimos que los resultados no mejoraban de forma sustancial y estábamos estancados en torno a 4.4 de Loss medio.

Para hacernos una idea más detallada de la precisión conseguida por el algoritmo, podemos obtener una métrica específica para algoritmos de detección de objetos llamada ***mAP***. ***mAP*** se refiere a *mean Average Precision* y es una medida de la *Precisión* y el *Recall* que va entre 0 y 1. Si recordamos, *Precision* mide como de precisas son nuestras predicciones, es decir, el número de predicciones bien realizadas. Por otro lado, *Recall* (o sensibilidad) mide, de todos los casos positivos, cuantos somos capaces de detectar correctamente.

Además, para cada predicción se calcula el IoU de la bounding box predicha con la bounding box real y normalmente consideramos que una predicción es correcta cuando IoU>=0.5 (TP). Si

IoU<0.5 es considerado un Falso Positivo (FP) y si es IoU>=0.5 pero la clase es incorrecta, se considera un Falso Negativo (FN). Con toda esa información, se procede a dibujar una gráfica de *Recall* frente a *Precision*. Antes de ello, es necesario corregir los valores mediante *Precisión Interpolada.* Básicamente para cada valor de *Recall* se coge la máxima *Precision* medida. Con ello obtendríamos el siguiente gráfico:



Normalmente se toman 11 puntos para calcular la precisión media (*Average Precision* (AP), en inglés) de forma que haciendo la media de todos ellos obtenemos el valor de *mAP*. Normalmente este proceso se hace para cada una de las clases a predecir y para todas en global. A continuación, podemos observar el resultado obtenido para nuestro entrenamiento en el conjunto de validación.

```
calculation mAP (mean average precision...
204
 detections_count = 28404, unique_truth_count = 13661
class_id = 0, name = globulo rojo, ap = 93.48%        (TP = 12530, FP
= 3991)
class_id = 1, name = trofozoito, ap = 60.55%     (TP = 155, FP = 84)
class_id = 2, name = dificil, ap = 30.77%        (TP = 49, FP = 86)
class_id = 3, name = aro, ap = 1.80%       (TP = 1, FP = 9)
class_id = 4, name = esquizonte, ap = 19.93%     (TP = 4, FP = 4)
class_id = 5, name = gametocito, ap = 11.42%     (TP = 4, FP = 18)
class_id = 6, name = leucocito, ap = 72.22%      (TP = 12, FP = 48)

 for thresh = 0.25, precision = 0.75, recall = 0.93, F1-score = 0.83
 for thresh = 0.25, TP = 12755, FP = 4240, FN = 906, average IoU =
65.66 %

 IoU threshold = 50 %, used Area-Under-Curve for each unique Recall
 mean average precision (mAP@0.50) = 0.414528, or 41.45 %
Total Detection Time: 25.000000 Seconds
```

Podemos ver como obtenemos para cada una de las clases, un rendimiento muy dispar propio de la naturaleza del dataset no balanceado. Vemos como el AP en la clase de glóbulo rojo es bastante elevado (93.48) pero en las clases más minoritarias el resultado es bastante bajo (gametocito ~ 11.42).

El ***mAP*** global obtenido es del 41.45 lo cual no es bastante pobre si lo comparamos con otros modelos similares para otros dataset y tenemos en cuenta la definición del propio problema.

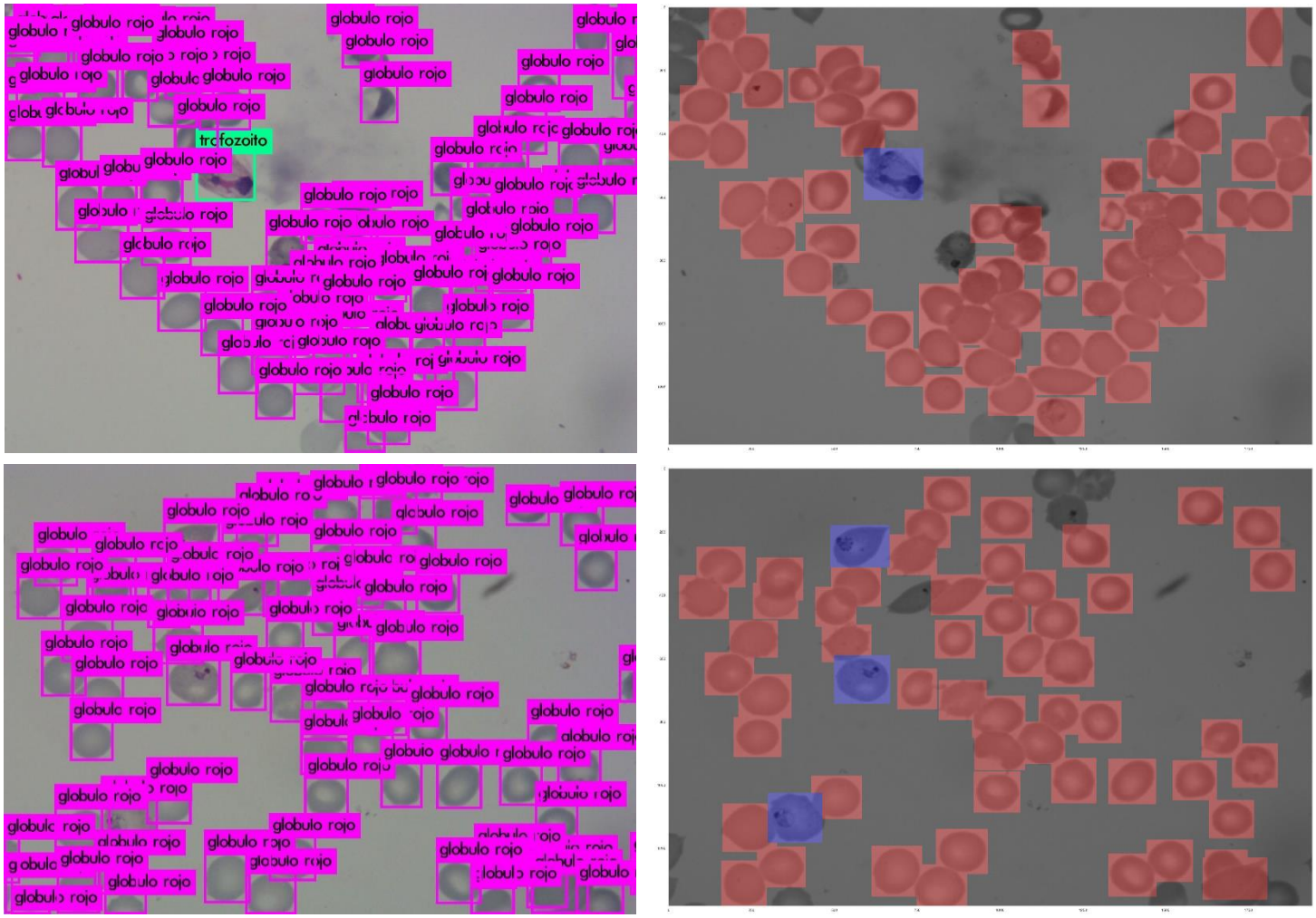
	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
Two-stage methods							
Faster R-CNN+++ [3]	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [6]	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [4]	Inception-ResNet-v2 [19]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [18]	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	**52.1**
One-stage methods							
YOLOv2 [13]	DarkNet-19 [13]	21.6	44.0	19.2	5.0	22.4	35.5
SSD513 [9, 2]	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513 [2]	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet [7]	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet [7]	ResNeXt-101-FPN	**40.8**	**61.1**	**44.1**	**24.1**	**44.2**	51.2
YOLOv3 608 × 608	Darknet-53	33.0	57.9	34.4	18.3	35.4	41.9

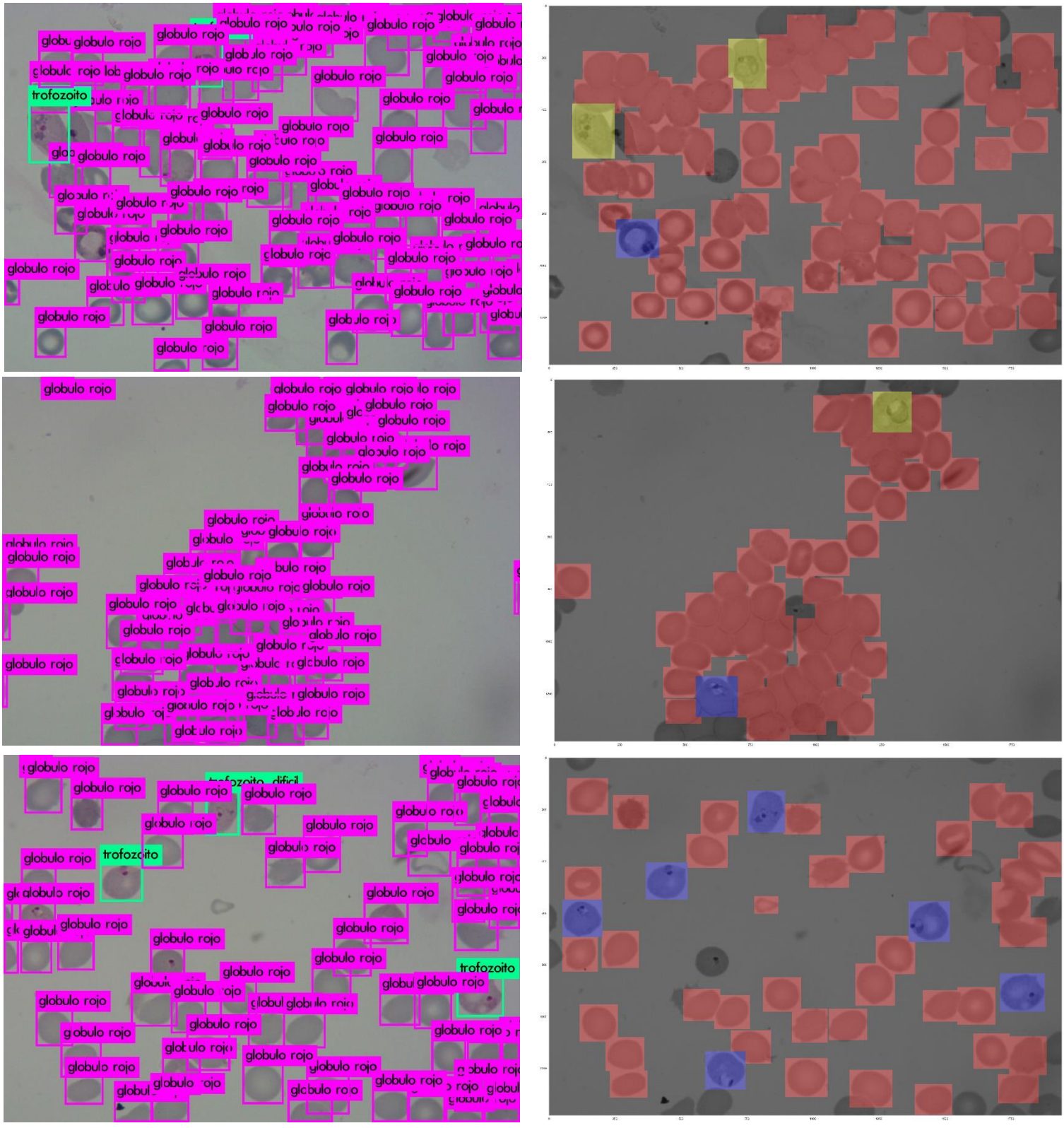
COCO for YOLOv3

En este caso observamos el dataset COCO (dataset utilizado para detección y segmentación con 80 clases diferentes) y vemos como el algoritmo YOLOv3 base obtiene unos valores de 57.9 para AP_{50} (el mismo utilizado en nuestra métrica) para dicho dataset y otras versiones de algoritmos de detección de objetos se mueven por esos valores.

Para visualizar los resultados, vamos a mostrar algunas imágenes del conjunto de test que el algoritmo no ha visto y compararla con la predicción obtenida por YOLO.

IMAGEN PREDICHA POR EL MODELO **IMAGEN REAL**





En general podemos observar como el modelo es capaz de detectar de forma precisa la mayoria de los globulos rojos incluso aquellos que no han sido etiquetados directamente. Sin embargo, y como era de esperar, a la hora de predicir ciertas clases, el modelo o no las clasifica todas correctamente o hay algunas que las confunde con la clase mayoritaria, los globulos rojos.

CONCLUSIONES Y MEJORAS

En este trabajo hemos visto en qué consiste el algoritmo YOLO y más concretamente su versión v3. Además, hemos entrenado un modelo ad-hoc para la detección de células infectadas de malaria con el fin de localizar en una imagen, las células tanto infectadas como no infectadas. Como concusiones al trabajo podemos destacar:

- La propia distribución del dataset no balanceado (95/5) hace muy difícil la obtención de buenos rendimientos sin un tratamiento previo de los datos. Aplicando técnicas para su corrección (SobreMuestreo o SubMuestreo) u obteniendo más imágenes de las clases positivas (minoritarias), el resultado probablemente mejoraría.
- La utilización de diferentes configuraciones dentro de la propia configuración del algoritmo YOLO (funciones de activación, dimensiones de los anchor boxes, thresholds utilizados, etc.) llevaría con toda seguridad a una mejora en el resultado final.
- El llevar a cabo el entrenamiento sobre imágenes lo más grandes posibles, aceleraría el proceso de llegar pronto a buenas soluciones. Sin embargo, el tiempo requerido de entrenamiento para llegar a un optimo global se ralentizaría demasiado.