



CENTRO UNIVERSITARIO
DE TECNOLOGÍA Y ARTE DIGITAL

Sistema de detección automática de baches en el asfalto a partir de imágenes

Diego Castro Viadero

Septiembre 2019

Tutor: Ricardo Moya García

Resumen

El estado del asfalto en carreteras tanto de ámbito nacional como de ámbito urbano es de alta importancia en relación a la seguridad vial. En la actualidad, gracias a los avances tecnológicos, se están desarrollando sistemas de detección automática de baches en el asfalto, permitiendo una detección precoz de estas irregularidades en la carretera.

Este proyecto pretende contribuir en este ámbito desarrollando un sistema de detección automática de baches a partir de imágenes. Partiendo de un conjunto de imágenes etiquetadas con baches, se ha entrenado una red neuronal *YOLO V3* y otra *YOLO V3 Tiny*. El entrenamiento se ha realizado con distintos tamaños de red y con distintos subconjuntos de imágenes, dando lugar a un total de quince modelos. Tras un estudio comparativo de las precisiones de los modelos, aquellos con mejores resultados han sido exportados y transformados para ser ejecutables en un dispositivo móvil. Finalmente se ha desarrollado una aplicación móvil Android que carga los modelos anteriormente exportados y los ejecuta utilizando como entrada la salida de la cámara del dispositivo.

Abstract

The state of asphalt on both national and urban roads is of high importance in relation to road safety. Currently, thanks to technological advances, automatic detection systems are being developed to detect potholes in the asphalt, allowing early detection of these irregularities on the road.

This project aims to contribute to this field by developing an automatic pothole detection system based on images. Starting from a set of images labeled with potholes, a neural network *YOLO V3* and another *YOLO V3 Tiny* have been trained. The training has been carried out with different network sizes and with different subsets of images, resulting in fifteen models. After a comparative study of the precisions of the models, those with better results have been exported and transformed to be executable in a mobile device. Finally, an Android mobile application has been developed that loads the previously exported models and executes them using the camera output of the device as input.

Agradecimientos

A mi tutor Ricardo Moya García, por su esfuerzo, dedicación, consejos y darme ánimos en todo momento.

A mi pareja Nerea, por toda la paciencia que ha tenido y por el continuo apoyo moral y ayuda que me ha brindado a lo largo de esta aventura.

A mi familia y amigos por los ánimos y por haber estado dispuestos a ayudarme en todo momento

Contenido

1. Introducción	5
1.1. Motivación	5
1.2. Objetivos	5
1.3. Estructura del trabajo	6
2. Estado del arte	7
3. Definición de requisitos y análisis	10
3.1. Definición de requisitos	10
3.2. Arquitectura	10
3.3. Tecnologías	11
4. Datos	12
4.1. Descripción de las fuentes de datos a utilizar	12
4.2. Estudio de los datos	12
4.3. Preprocesamiento de las imágenes	14
5. Técnicas de Deep Learning y métodos de evaluación	16
5.1. Explicar las técnicas de DL que se van a utilizar en el proyecto . .	16
5.2. Explicar los métodos de evaluación que se van a utilizar en el proyecto	19
6. Implementación y evaluación de las técnicas	23
6.1. Detalles de la implementación de las técnicas de DL aplicadas . .	23
6.2. Evaluación de las técnicas	31
7. Resultados	34
7.1. Resultados del proyecto	34
8. Conclusiones	46
8.1. Evaluación del proyecto	46
8.2. Alternativas y posibles mejoras que podrían haberse aplicado al proyecto (trabajos futuros)	47
8.3. Conclusiones personales	48

1. Introducción

1.1. Motivación

En los últimos años las prioridades en relación a la seguridad vial han provocado un cambio de mentalidad en la sociedad. Muchos de los últimos avances tecnológicos están orientados al desarrollo de medios de transporte más seguros. Uno de estos avances que contribuye a la mejora de la seguridad es la inclusión de un sistema de detección de desperfectos en la calzada.

En la actualidad, en el ayuntamiento de Madrid, existe un sistema de sugerencias y reclamaciones [1] que permite al ciudadano, entre otras opciones, denunciar la existencia de irregularidades en el asfalto, tener un registro de las mismas y planificar su subsanación. Según el último informe publicado de sugerencias y reclamaciones del ayuntamiento de Madrid [2], en el primer semestre del año 2018, se registraron 3.000 reclamaciones en materia de *vías y espacios públicos* de las cuales el 50 % correspondieron a la submateria *aceras y calzadas*.

Este sistema de funcionamiento actual, que delega en el ciudadano la tarea de reporte de este tipo de desperfectos, dificulta y retrasa la puesta en conocimiento de las irregularidades a las autoridades responsables aumentando la probabilidad de que suceda algún incidente.

Algunas marcas de vehículos han desarrollado innovaciones, que son capaces de detectar baches cuando pasan sobre ellos y adaptar la dureza de la suspensión para conseguir una conducción más segura y cómoda. También contemplan un envío sistemático de la detección de baches, en tiempo real, tanto a otros vehículos como a las autoridades pertinentes. Otras marcas plantean la inclusión de cámaras que permitan la detección del bache sin necesidad de pasar por encima de este.

Con estas innovaciones las autoridades recuperan el control sobre la detección de baches, se libera al ciudadano de esta tarea y se reduce la probabilidad de incidentes gracias a que se dispone de la información con más antelación.

1.2. Objetivos

Con este proyecto se pretende realizar un desarrollo que tenga una utilidad social, y que al mismo tiempo permita ampliar los conocimientos adquiridos durante el máster de Big Data & Data Science, impartido por la U-TAD, en el ámbito del procesamiento de imágenes, concretamente en la detección de objetos en imágenes.

En este trabajo se desarrollará un sistema de detección de baches en tiempo real, en línea con los últimos avances tecnológicos. El objetivo es entrenar una red neuronal con un conjunto de imágenes, en las que los baches han sido etiquetados previamente, y obtener como resultado un modelo exportable para ser ejecutado en un dispositivo móvil.

1.3. Estructura del trabajo

En la sección *Estado del arte* se explica qué técnicas existen hoy en día para solucionar el problema de detección de objetos, centrándose en la resolución del mismo mediante el uso de redes neuronales.

En la sección *Definición de requisitos y análisis* se exponen los requisitos del proyecto y se muestra la arquitectura utilizada para su implementación.

La sección *Datos* describe el conjunto de datos utilizado. También muestra el análisis exploratorio que se ha realizado sobre el mismo y el tratamiento aplicado sobre los datos antes de ser utilizados.

En la sección *Técnicas de Deep Learning y métodos de evaluación* se describen a nivel teórico las principales técnicas y conceptos aplicados en el trabajo, así como las métricas que se utilizan para evaluar los resultados obtenidos.

La sección *Implementación y evaluación de las técnicas* describe, desde un nivel técnico, cómo se ha implementado la solución y muestra la evaluación de los resultados obtenidos.

En la sección *Resultados* se muestran algunos ejemplos representativos concretos.

La sección *Conclusiones* contiene una valoración del trabajo realizado y se comentan posibles mejoras a realizar para continuar con el desarrollo del mismo.

2. Estado del arte

La detección de objetos en imágenes es un problema que presenta múltiples retos. El primero de ellos es que las imágenes no se centran en un único objeto, sino que en una misma imagen puede haber múltiples objetos a detectar y además tratarse de objetos de distintos tipos. El segundo de los retos es el tamaño de los objetos a identificar, que puede ser variable. Y el tercero de los retos es que se están resolviendo dos problemas al mismo tiempo: localizar objetos en una imagen y clasificar los objetos localizados.

Para resolver los problemas de detección de objetos existen dos aproximaciones. La primera de las aproximaciones es una aproximación clásica, basada en técnicas de machine learning. Un ejemplo es *Viola-Jones*, que está basado en clasificadores binarios y que se ha usado en las cámaras de fotos para la detección de caras.

La segunda aproximación es el uso del deep learning, lo cual ha supuesto una revolución y ha cambiado las reglas del juego. Esta aproximación para la resolución de este tipo de problemas es relativamente reciente y ha estado en constante evolución.

En estos últimos años han habido múltiples desarrollos para afrontar el problema de detección de objetos con deep learning. A continuación se va a hacer un repaso de los más relevantes [3] [4] [5] [6] [7].

R-CNN

Una de las primeras soluciones que hicieron uso de técnicas de deep learning para la resolución de problemas de detección de objetos fue *R-CNN (Region-based Convolutional Neural Networks)* [8]. El funcionamiento de R-CNN se resume en tres pasos:

1. Se escanea la imagen en busca de posibles objetos. Mediante un algoritmo de proposición de regiones, el más habitual es la *búsqueda selectiva (selective search)* [9], se obtienen una serie de regiones candidatas de contener un objeto, denominadas *RoI (Region of Interest)*. Se obtienen aproximadamente unas 2000 RoIs.
2. Se ejecuta una red neuronal convolucional para extraer las características (*features*) de cada una de las regiones de interés
3. Las características obtenidas de la red neuronal convolucional alimentan:
 - a) un SVM para clasificar el objeto
 - b) un regresor lineal para ajustar la región candidata al objeto

Aunque con esta aproximación se obtienen buenos resultados, presenta muchos inconvenientes, entre los cuales destaca la dificultad para entrenar. Por un lado hay que obtener las regiones de interés del conjunto de entrenamiento, después

hay que ejecutar la red neuronal convolucional sobre cada una de las regiones candidatas para obtener las características. Finalmente hay que entrenar el clasificador SVM y el regresor lineal con las características obtenidas.

Fast R-CNN

R-CNN no tardó en evolucionar hacia una solución de deep learning más pura. El mismo autor de R-CNN fue el autor de su evolución *Fast R-CNN* [10].

En este caso se ejecuta la red neuronal convolucional sobre la imagen completa para obtener las características. Una vez obtenidas las características se aplica la *búsqueda selectiva* para obtener las regiones de interés. A continuación se reduce el número de regiones de interés con una capa *RoI Pooling* y una red neuronal totalmente conectada. Por último para realizar la clasificación de los objetos se utiliza un clasificador *softmax*, en lugar de SVM. Para ajustar las regiones se sigue utilizando una regresión lineal.

Esta aproximación presenta múltiples ventajas con respecto a su predecesora. Por un lado, la red neuronal convolucional se ejecuta una única vez, en lugar de una vez por cada región de interés (aproximadamente unas 2000 regiones de interés). Por otro lado, en vez de utilizar múltiples SVM para realizar la clasificación, se utiliza un único clasificador softmax. Con todo esto se mejora mucho el rendimiento, además de que se facilita el entrenamiento. Sin embargo, sigue teniendo una pega, que es el uso de la búsqueda selectiva para obtener las regiones de interés.

Faster R-CNN

La aproximación Fast R-CNN sufrió una tercera evolución para dar lugar a *Faster R-CNN* [11]. En esta tercera iteración se suprimió el uso de la *búsqueda selectiva* y se introdujo en su lugar lo que se denominó *red de proposición de regiones (RPN, Region Proposal Network)*. De esta forma se consiguió mejorar bastante el rendimiento y tener una aproximación que es entrenable de principio a fin.

YOLO

Hasta la aparición de *YOLO*, los enfoques que se habían utilizado consistían en adaptar las técnicas de clasificación de imágenes para la detección de objetos. YOLO, que responde al acrónimo *You Only Look Once*, cambia el enfoque y afronta el problema de detección como una regresión. Con una única red neuronal convolucional, y de una vez, es capaz de predecir tanto las regiones como las clases de los objetos. Esta simplicidad le permite realizar predicciones en tiempo real.

Con YOLO se divide la imagen en una rejilla de dimensiones $S \times S$. En cada una de las celdas de la rejilla se obtienen B regiones candidatas de contener

un objeto con un valor de confianza. Esta confianza refleja la probabilidad de contener un objeto, y cómo de ajustada es la región candidata con respecto a la región a detectar. Esto último se hace en base al *IoU (Intersection over Union)*, de la región candidata con respecto a la verdadera.

El diseño original, inspirado en *GoogLeNet*, está formado por 24 capas convolucionales y 2 capas totalmente conectadas. Con respecto a los resultados, está un poco por debajo de Faster R-CNN, en lo que respecta a precisión, pero a cambio ofrece un alto rendimiento.

Al igual que sucedió con R-CNN, YOLO ha estado en continua evolución. La última versión desarrollada es la versión *YOLO V3* [12]. En esta última iteración se han hecho pequeñas mejoras y se ha hecho la red más grande (106 capas convolucionales), mejorando la detección de objetos pequeños, a cambio de hacerla un poco más lenta con respecto a la anterior.

Más ...

Existen otras muchas aproximaciones, como por ejemplo: *SSD*, *R-FCN*, *Mask R-CNN*, etc., en las que no se va a entrar en detalle, ya que sería muy difícil abarcar todas.

3. Definición de requisitos y análisis

3.1. Definición de requisitos

A continuación se enumeran los requisitos para el desarrollo del trabajo:

- Implementar y entrenar una red neuronal que genere un modelo capaz de detectar baches en el asfalto a partir de imágenes
- El modelo deberá ser capaz de procesar una imagen en un tiempo lo más cercano posible a 1/30s, para ser capaz de procesar video en tiempo real
- El modelo deberá poderse ejecutar en un dispositivo móvil Android
- El modelo será alimentado directamente con la salida de la cámara del dispositivo móvil

3.2. Arquitectura

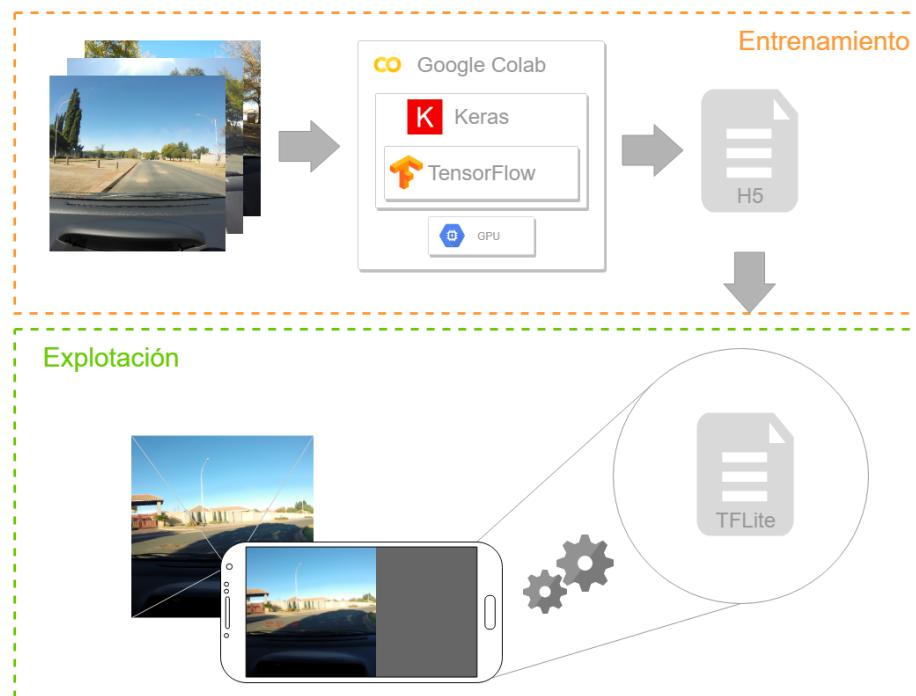


Figura 1: Arquitectura de la solución

Como se puede observar en la figura 1, la arquitectura está dividida en dos partes. Una parte se utiliza para entrenar la red neuronal y generar el modelo. La otra parte se utiliza para explotar el modelo generado.

La parte utilizada durante el entrenamiento requiere de un hardware potente y del uso de GPU para reducir los tiempos de entrenamiento. La explotación del modelo, al contrario de lo que sucede con el entrenamiento, no requiere de un hardware potente y se ejecuta directamente en un dispositivo móvil en posesión del usuario final. El modelo obtenido después de la fase de entrenamiento, se transforma a un formato que está pensado para ser ejecutado en dispositivos con recursos limitados.

3.3. Tecnologías

Todo el proyecto ha sido desarrollado utilizando Python, salvo la aplicación Android, que ha sido desarrollada en Java.

Para el procesamiento de imágenes se ha utilizado el paquete Python OpenCV. Este procesamiento incluye la lectura de imágenes en formato jpg, recorte, reescalado y volteo de las imágenes, visualización de las predicciones obtenidas, etc.

Para la implementación de la red neuronal se ha utilizado Keras, que es un envoltorio sobre Tensorflow que simplifica su uso. Dada una arquitectura de red neuronal, con Keras es muy sencillo definir las capas y sus interconexiones. Además Keras proporciona clases que facilitan la definición de un conjunto de imágenes como entrada de la red neuronal. Keras también proporciona una serie de eventos con la idea de poder suscribirse a los mismos y poder reaccionar en consecuencia, como por ejemplo, suscribirse al evento de final de época y salvar el modelo si ha mejorado con respecto a la época anterior.

Las redes neuronales utilizadas en el proyecto no se han implementado de cero, sino que se ha partido de dos implementaciones en Keras de *YOLO v3* [13] y *YOLO v3 Tiny* [14]. Se han unido ambas implementaciones en una única [15] que soporta ambos tipos de red YOLO. También se han realizado múltiples desarrollos para mejorar la funcionalidad, como por ejemplo: soporte en formato txt de las etiquetas de las imágenes, nuevos parámetros de configuración para mejorar el resultado del entrenamiento, etc.

Para la ejecución de los modelos obtenidos en un dispositivo móvil se ha utilizado TFLite. Este paquete permite transformar distintos tipos de modelo (keras, tensorflow) a formato TFLite y ejecutar estos modelos en dispositivos con recursos reducidos. En las últimas versiones de esta librería se soporta, aunque de forma experimental, el uso de la GPU del dispositivo.

La aplicación móvil ha sido desarrollada en java para la plataforma Android. TFLite dispone de una librería java que simplifica la carga del modelo y su ejecución.

4. Datos

4.1. Descripción de las fuentes de datos a utilizar

El juego de datos ha sido obtenido de kaggle [16] y se compone de un total de 1900 imágenes, tomadas desde el interior de un coche, con un tamaño igual a 3680x2760 píxeles (formato 4:3), y de un conjunto de ficheros de texto con el etiquetado de las mismas. Las imágenes se dividen en dos subconjuntos: uno de 1297 imágenes para el entrenamiento y otro de 603 imágenes para la evaluación del modelo. Por cada uno de los subconjuntos de imágenes existe un fichero de texto con el etiquetado de las mismas. Cada una de las líneas del los ficheros de texto contiene las etiquetas de una imagen. La estructura de cada línea es la siguiente:

```
<RUTA_IMG> <NUMERO_DE_ETIQUETAS>(<X0> <Y0> <ANCHO> <ALTO>)+
```

Para facilitar el posterior tratamiento, se ha realizado una transformación del formato de los ficheros de etiquetas al siguiente formato:

```
<RUTA_IMG>(<X0>, <Y0>, <ANCHO>, <ALTO>, <CLASE>)+
```

4.2. Estudio de los datos

En una fase inicial se ha realizado un análisis del tamaño de los baches con respecto al tamaño de la imagen. Esto es un aspecto importante a tener en cuenta de cara a determinar el algoritmo a utilizar para la detección de objetos. Los algoritmos de detección de objetos, en general se comportan peor cuanto más pequeños son los objetos a detectar.

Como se observa en la figura 2, la mayoría de los baches tienen una anchura inferior a 200 píxeles y una altura inferior a 50 píxeles. Este factor será tenido en cuenta en el preprocesamiento de las imágenes.

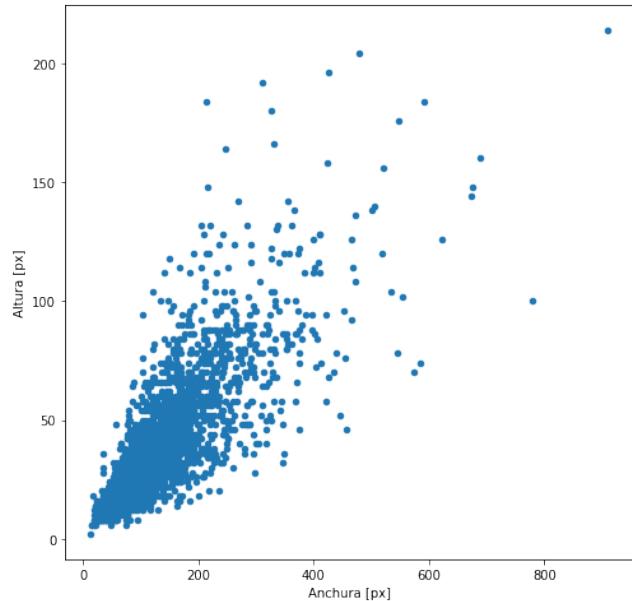


Figura 2: Tamaños de los baches en píxeles

También se ha realizado un estudio de la localización de los baches en las imágenes. Tal y como se ve en la figura 3, los baches están localizados principalmente en el centro de la imagen. La parte inferior se corresponde con el salpicadero del coche y la parte superior se corresponde con paisaje.

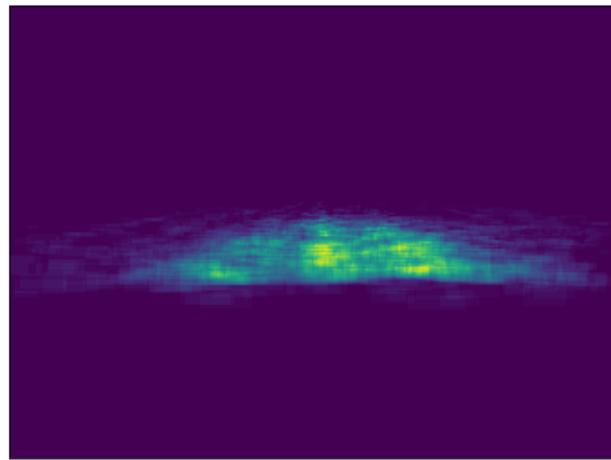


Figura 3: Localizaciones de los baches en las imágenes

4.3. Preprocesamiento de las imágenes

Tanto en la fase de entrenamiento, como para hacer una predicción, las imágenes van a ser redimensionadas al tamaño de la red neuronal, la cual tiene una relación de aspecto 1:1. Para redimensionar una imagen con una relación de aspecto 4:3, y al mismo tiempo, transformarla en una imagen con relación de aspecto 1:1, lo que se hace es redimensionar el lado más grande de la imagen manteniendo la relación de aspecto, es decir, aplicando el mismo factor de redimensionamiento al lado más pequeño. Una vez redimensionada, se rellena con gris la zona superior y la zona inferior de la imagen para cuadrarla. En la figura 4 se muestra un ejemplo gráfico.

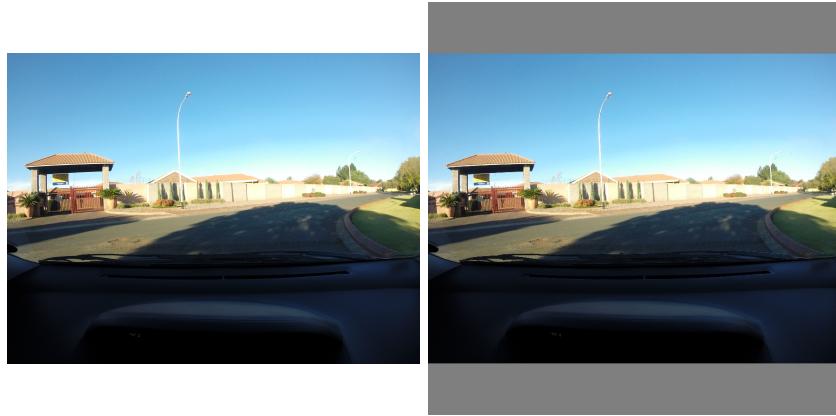


Figura 4: A la izquierda la imagen original redimensionada a tamaño 920x690 px (manteniendo la relación de aspecto 4:3). A la derecha la imagen redimensionada con el relleno para que tenga una relación de aspecto 1:1 (920x920 px)

El redimensionamiento se hace en base al lado más grande de la imagen, que en el ejemplo anterior es la anchura. Para determinar el factor de redimensionamiento, se divide la anchura la imagen final entre la anchura de la imagen original, en este caso: $920/3680 = 0,25$. A continuación, se aplica este factor de redimensionamiento a ambos lados de la imagen, resultando en un tamaño de 920x690 píxeles. Por último se calcula el relleno que haría falta a cada lado de la imagen: $(920 - 690)/2 = 115$.

Siguiendo con este ejemplo, si en la imagen original hubiese un bache de tamaño 160x24 píxeles, y se aplicase este factor de redimensionamiento, el bache redimensionado tendría unas dimensiones de 40x6 píxeles, lo cual sería un tamaño bastante pequeño ya que únicamente tiene 6 píxeles de alto (de 920 que tiene la imagen).

Sin embargo, si previo al redimensionamiento de la imagen, se recortan los extremos izquierdo y derecho de la imagen, de tal forma que tenga una relación de aspecto 1:1, se consigue que el factor de redimensionamiento sea mayor y que

por tanto los baches redimensionados sean también más grandes. Esta técnica tiene un inconveniente, y es que la imagen original se está recortando, por lo que está habiendo una pérdida de información. Este inconveniente no es un impedimento, ya que en el apartado 4.3 se ha comprobado que la mayor parte de los baches están en el centro de las imágenes, y que recortando los extremos de las mismas la pérdida de información es mínima.

Para aplicar esta técnica, en primer lugar habría que calcular los recortes que hay que hacer a cada lado de la imagen original. Para ello se calcula la diferencia entre la anchura y la altura de la imagen y se divide por dos: $(3680 - 2760)/2 = 460$. Una vez recortada la imagen se calcula el factor de redimensionamiento: $920/2760 = 0,333$. Por último se aplicaría este factor de redimensionamiento a la altura y la anchura de la imagen.

Si aplicamos este nuevo factor de redimensionamiento al tamaño del bache del ejemplo anterior (160x24 píxeles), el tamaño del bache redimensionado sería 53x8 (un 75 % más grande).

5. Técnicas de Deep Learning y métodos de evaluación

5.1. Explicar las técnicas de DL que se van a utilizar en el proyecto

A continuación se explicarán las principales técnicas utilizadas para el desarrollo del proyecto. Algunas de las técnicas mencionada no son específicas del deep learning.

La primera de las técnicas es una de las técnicas *generalistas*, propia de los algoritmos de aprendizaje en general. Se trata del *hold-out*, que consiste en dividir el conjunto de datos en dos subconjuntos: uno que se utiliza para el entrenamiento y otro que se utiliza para evaluar el modelo obtenido tras el entrenamiento.

Para el desarrollo del proyecto se ha implementado una *red neuronal convolucional (CNN)* [17] [18] [19]. Las *CNNs* están compuestas por una secuencia de capas convolucionales, para la extracción de características, seguidas de unas capas perceptrón simples, para la clasificación final. Las capas convolucionales forman una jerarquía, de forma que las capas del principio son capaces de detectar formas básicas y cuanto más “profunda” es la capa, más complejas son las formas que son capaces de detectar. Por ejemplo: una capa inicial sería capaz de detectar líneas y círculos, mientras que una capa profunda sería capaz de detectar una cara.

Las capas convolucionales están compuestas por neuronas que realizan la operación de *convolución*. Dichas neuronas reciben en la entrada una matriz bidimensional, que se corresponde con la imagen, si se trata de la primera capa de la red, o con la salida de la capa convolucional anterior. Además, cada una de estas neuronas tiene definido un *kernel*, que es otra matriz bidimensional de dimensiones reducidas.

Entrada	Kernel
0 1 1 1 0	0 1 0
0 1 1 1 0	0 1 0
1 1 1 1 1	0 1 0
0 1 1 1 0	0 1 0
0 1 1 1 0	0 1 0

Figura 5: Ejemplo de neurona convolucional con una entrada de 5x5 y un kernel de 3x3

La operación de *convolución* consiste en hacer un barrido del *kernel* por la matriz de entrada, de izquierda a derecha y de arriba a abajo. En cada una de las posiciones del barrido se aplica el producto escalar entre el kernel y la submatriz de la matriz de entrada sobre la que se encuentre. En la figura 6 se puede ver gráficamente el proceso.

Entrada	Kernel
0 1 1 1 0	0 1 0
0 1 1 1 0	0 1 0
1 1 1 1 1	0 1 0
0 1 1 0	0 1 0
0 1 1 0	0 1 0

Figura 6: Representación gráfica de operación convolucional

Para realizar el producto escalar, las matrices se *aplanan*, de tal forma que la submatriz de entrada quedaría de la siguiente forma:

$$\begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \Rightarrow [0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1]$$

El kernel:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \Rightarrow [0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0]$$

Y por lo tanto el producto escalar:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = 3$$

Cada una de las neuronas de las capas convolucionales no tiene un único kernel asociado, sino que tienen varios. Teniendo en cuenta esto y suponiendo que tenemos como entrada una imagen en blanco y negro de $28x28$ píxeles y que cada neurona tiene 32 kernels de $3x3$, la primera capa convolucional estaría compuesta por $28x28 = 784$ neuronas, y se obtendrían como resultado $28x28x32 = 25,088$ matrices de salida, que se correspondería con el número de neuronas necesarias para la siguiente capa convolucional. Esto hace necesario introducir un mecanismo de reducción para que el número de neuronas necesarias no crezca tan desmesuradamente con cada capa. Este mecanismo se conoce como *pooling* y existen distintos tipos de pooling. Uno de ellos es el *MaxPooling*, y consiste en deslizar una ventana bidimensional sobre la matriz de entrada, de izquierda a derecha y de arriba a abajo. En cada una de las posiciones se tomará como valor resultante el máximo valor presente en la ventana. La capacidad de reducción del MaxPooling depende de dos factores: el tamaño de la ventana y el tamaño del salto que se da hacia la derecha y hacia abajo. En la figura 7 se muestra un ejemplo gráfico.

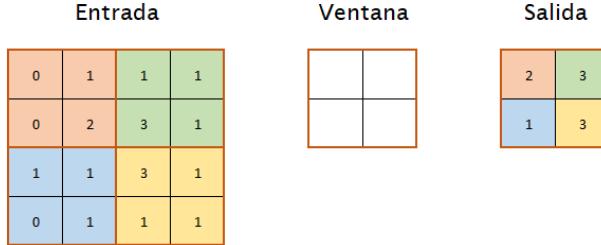


Figura 7: Representación gráfica de MaxPooling con una entrada de 4x4, una ventana de 2x2 y un salto de 2

Las redes neuronales convolucionales utilizadas para la detección de objetos tienen como entrada una matriz bidimensional que se corresponde con la imagen (en caso de ser una imagen a color, serán 3 matrices bidimensionales, una por cada canal de color). Como salida se obtiene una lista de predicciones, cada una de las cuales se compone de una región representada por cuatro variables (coordenadas x e y de la esquina superior izquierda, anchura y altura) y de las probabilidades pertenencia del objeto en la región a cada una de las clases que la red es capaz de predecir.

Otra de las técnicas que se han utilizado en el proyecto es la denominada *transferencia del conocimiento (transfer learning)* [20]. Entrenar un modelo de detección de objetos es un proceso muy costoso, ya que se necesita un gran volumen de imágenes y una gran capacidad de cómputo. El *transfer learning* viene a palear este problema ya que permite utilizar un modelo ya entrenado como punto de partida para entrenar otro. Como se ha comentado anteriormente, las primeras capas convolucionales son capaces de detectar formas básicas y a medida que se profundiza en las capas, estas detectan formas más específicas. Si ya tenemos entrenado un modelo (a) para detectar un determinado tipo de objetos y queremos entrenar otro (b) para detectar otro tipo de objetos, las primeras capas de ambos modelos van a detectar el mismo tipo de formas, y serán las capas más profundas las que se diferencien. Por lo tanto podríamos utilizar el modelo *a* como punto de partida para el modelo *b*, y este último podrá aprender más rápido ya que las primeras capas ya están entrenadas para detectar formas básicas.

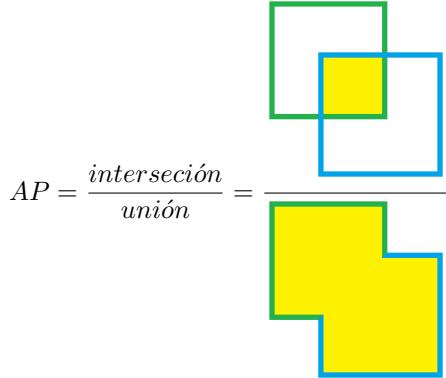
5.2. Explicar los métodos de evaluación que se van a utilizar en el proyecto

La métrica que se ha utilizado para evaluar el modelo obtenido es la *AP* (Average Precision), que es la métrica que se utiliza para evaluar modelos de detección de objetos.

Antes de explicar en qué consiste la métrica *AP* hay que explicar una serie de conceptos en los cuales está basada: IoU (intersección sobre la unión), precisión

(precision) y sensibilidad (recall).

El concepto de *IoU* mide cuánto se solapan dos regiones: la predicha y la que debería ser detectada. Se calcula dividiendo la región obtenida mediante la intersección de la región predicha y la región a detectar entre la región obtenida mediante la unión de ambas regiones.



La *precisión* (precision) mide la capacidad del modelo para detectar únicamente los objetos relevantes. Se calcula como el porcentaje de predicciones positivas acertadas frente a todas las predicciones positivas predichas:

$$\text{precisión} = \frac{TP}{TP + FP} = \frac{TP}{\text{todas las predicciones positivas}}$$

La *sensibilidad* (recall) mide la capacidad del modelo para detectar todos los objetos relevantes. Se calcula como el porcentaje de predicciones positivas acertadas frente a todas las existentes:

$$\text{sensibilidad} = \frac{TP}{TP + FN} = \frac{TP}{\text{todas las regiones a detectar}}$$

Tanto en el cálculo de la *precisión* como en el cálculo de la *sensibilidad*, para determinar si una predicción es positiva, se utiliza el *IoU*. Se define un umbral para el *IoU* (normalmente suele ser 0.5) y si se supera dicho umbral, la predicción es considerada una predicción positiva.

La métrica *AP* se calcula como el área debajo de la curva *precisión-sensibilidad* (precision-recall). En el eje de las abscisas se representa la *sensibilidad* (recall) y en el eje de las ordenadas se representa la *precisión* (precision).

A continuación se va a mostrar un ejemplo práctico de cómo se calcula la *AP*. Para este ejemplo se dispone de una serie de imágenes con un total de 4 baches a detectar. En la tabla 1 se puede ver el cálculo de la *precisión* y de la *sensibilidad* para las predicciones obtenidas. La columna *Positivo* indica si la predicción es

positiva, es decir, si el valor de *IoU* supera el umbral definido, que en este caso es 0.5. Las columnas *TP* y *FP* muestran el acumulado de sus respectivos valores.

IoU	Positivo	TP	FP	Precisión	Sensibilidad
0.912933	1	1	0	1.000000	0.25
0.711111	1	2	0	1.000000	0.50
0.387983	0	2	1	0.666667	0.50
0.387983	0	2	2	0.500000	0.50
0.387983	0	2	3	0.400000	0.50
1.000000	1	3	3	0.500000	0.75
0.225986	0	3	4	0.428571	0.75
0.225986	0	3	5	0.375000	0.75
1.000000	1	4	5	0.444444	1.00

Tabla 1: Cálculo de la precisión y sensibilidad para las predicciones

Una vez se tienen calculados los valores de *precisión* y de *sensibilidad* se calcula la curva *precisión-sensibilidad* como se puede ver en la figura 8. Para realizar el cálculo del área debajo de la curva se realiza un suavizado de la misma. Este suavizado consiste en establecer como valor de *precisión* para un determinado valor de *sensibilidad*, el valor de *precisión* más alto que se encuentre a su derecha. Por ejemplo, para la *sensibilidad* 0.6 se establece como valor de *precisión* el valor más alto a su derecha, que en este caso es 0.5. En color naranja se puede ver la curva suavizada.

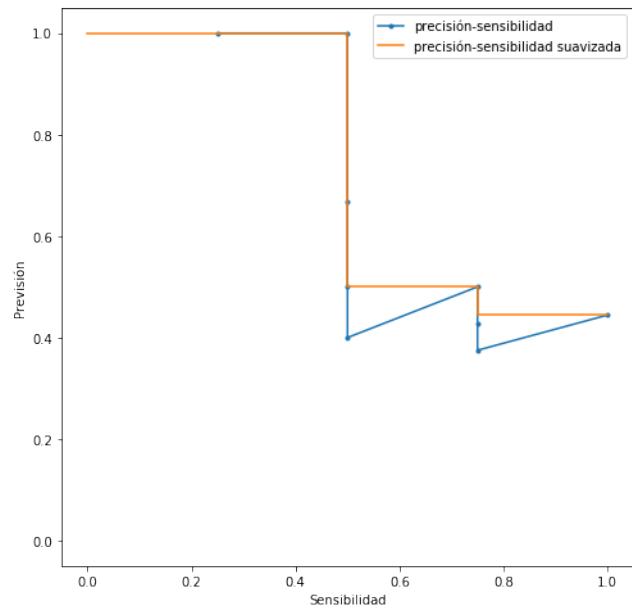


Figura 8: Curva precisión-sensibilidad

Por lo que finalmente, para este ejemplo, el cálculo del AP sería:

$$AP = (0,5 - 0) \cdot 1 + (0,75 - 0,5) \cdot 0,5 + (1 - 0,75) \cdot 0,44 = 0,5 + 0,125 + 0,11 = \mathbf{0.735}$$

6. Implementación y evaluación de las técnicas

6.1. Detalles de la implementación de las técnicas de DL aplicadas

En este apartado se explica cómo se ha implementado el proyecto. Para ello se muestra una visión general de la estructura del código, se detallan y explican las distintas opciones de configuración para su ejecución y se explica un flujo completo de principio a fin: entrenamiento, evaluación, predicción, transformación y explotación en dispositivo móvil.

Estructura del proyecto

Como se ha comentado anteriormente el proyecto se basa en dos implementaciones de YOLO (*v3* y *v3 tiny*) utilizando keras. Ambas implementaciones han sido unificadas en una [15] que soporta ambos tipos de red. Dicha implementación presenta la estructura que se muestra a continuación (únicamente se muestran los elementos más relevantes y con un asterisco los scripts principales, el resto son auxiliares):

- utils
 - bbox.py
 - utils.py
- annotations.py
- callbacks.py
- evaluate.py (*)
- predict.py (*)
- train.py (*)
- yolo_generator.py
- yolo_tiny_generator.py
- yolo_tiny_weight_reader.py
- yolo_tiny.py
- yolo_v3_weight_reader.py
- yolo.py

A continuación se describe la funcionalidad general de cada uno de estos scripts:

- **utils/bbox.py**: contiene la definición de la clase `BoundBox` que se utiliza para representar cada una de las predicciones obtenidas con el modelo. Tiene los atributos necesarios para identificar la región y la probabilidad de pertenencia del objeto a cada una de las clases.

- `utils/utils.py`: contiene funciones de cálculo auxiliares, como por ejemplo una función para calcular el *AP* de un modelo, otra función para el procesamiento de la salida del modelo, etc.
- `annotations.py`: este script permite procesar los ficheros con las etiquetas de las imágenes. Es capaz de procesar annotaciones en formato `voc` y en formato `txt`.
- `callbacks.py`: contiene la definición de *callbacks* de keras customizados. Un ejemplo, es un callback que se ha definido para que se ejecute al final de cada época y persista en disco el modelo actual siempre que mejore con respecto al de la época anterior.
- `evaluate.py`: este script permite evaluar un modelo calculando su *AP*.
- `predict.py`: este script permite obtener las predicciones de una imagen, de las imágenes de un directorio o de un video.
- `train.py`: este script permite entrenar un modelo en base a un fichero de configuración, recibido como argumento.
- `yolo_generator.py`: contiene la definición de la clase `BatchGenerator` que se utiliza para alimentar los modelos de tipo *YOLO v3*. Tiene configurado, entre otras cosas, un directorio que contiene imágenes y sus correspondientes anotaciones. Se encarga de proporcionar al modelo las imágenes a medida que las va necesitando. Es muy útil cuando se tienen grandes cantidades de datos, ya que evita tener que cargar todos los datos en memoria. Se encarga también del preprocesamiento de las imágenes, que en este caso consiste en redimensionar la imagen al tamaño de la red neuronal y normalizarla dividiendo entre 255. Además, durante la fase de entrenamiento, realiza otro tipo de transformaciones aleatorias, como por ejemplo rotaciones, cambios de saturación o de color.
- `yolo_v3_weight_reader.py`: contiene la definición de la clase `WeightReader` que es capaz de transformar los pesos ya entrenados de la red *YOLO v3* en formato `darknet` a formato `keras`.
- `yolo.py`: contiene la definición del modelo *YOLO v3* con cada una de sus capas convolucionales y sus interconexiones.
- `yolo_tiny_generator.py`: contiene la definición de la clase `BatchGenerator`, pero adaptada para los modelos *YOLO v3 tiny*.
- `yolo_tiny_weight_reader.py`: contiene la definición de la clase `WeightReader` que es capaz de transformar los pesos ya entrenados de la red *YOLO v3 tiny* en formato `darknet` a formato `keras`.
- `yolo_tiny.py`: contiene la definición del modelo *YOLO v3 tiny* con cada una de sus capas convolucionales y sus interconexiones.

Configuración

Tanto el entrenamiento, como la evaluación, como la predicción de imágenes se basan en un fichero de configuración que tiene el siguiente formato:

```
{  
    "model": {  
        "type": "tiny",  
        "min_input_size": 256,  
        "max_input_size": 256,  
        "anchors": [ 17,6, 21,9, 24,6, 27,8, 33,11, 36,17 ],  
        "labels": [ "pothole" ],  
        "data_load_method": "txt"  
    },  
    "train": {  
        "train_image_folder": "/dataset/train",  
        "train_annot": "/dataset/train_annotations.txt",  
        "cache_name": "/dataset/train_cache.pickle",  
        "train_times": 4,  
        "batch_size": 4,  
        "learning_rate": 0.0001,  
        "nb_epochs": 20,  
        "warmup_epochs": 3,  
        "ignore_thresh": 0.5,  
        "early_stopping_patience": 12,  
        "reduce_lr_on_plateau_patience": 4,  
        "gpus": "0",  
        "grid_scales": [ 1, 1 ],  
        "obj_scale": 5,  
        "noobj_scale": 1,  
        "xywh_scale": 1,  
        "class_scale": 1,  
        "tensorboard_dir": "logs/",  
        "saved_weights_name": "/model/v1.h5",  
        "pretrained_weights": "/model/yolo_v3_tiny_weights.h5",  
        "debug": true  
    },  
    "valid": {  
        "valid_image_folder": "/dataset/test",  
        "valid_annot": "/dataset/test_annotations.txt",  
        "cache_name": "/dataset/test_cache.pickle",  
        "duplicate_thresh": 0.20,  
        "valid_times": 1  
    }  
}
```

Código 1: Configuración de ejemplo

La configuración está dividida en tres secciones. La sección `model` donde se encuentran configuraciones generales del modelo. La sección `train` donde se encuentran las configuraciones para la fase de entrenamiento. Y por último la sección `valid` donde se encuentran las configuraciones para la evaluación del modelo entrenado.

A continuación se describen las principales propiedades de configuración:

- `model.type`: tipo de modelo que se quiere entrenar. Se soportan dos tipos: `v3` y `tiny`, que se corresponden con *YOLO v3* y *YOLO v3 tiny* respectivamente.
- `model.min_input_size` y `model.max_input_size`: tamaño mínimo y máximo de la red. La red neuronal YOLO en su versión 3 es una red neuronal de tamaño variable. Por cada 10 imágenes procesadas, durante la fase de entrenamiento, se hará un redimensionamiento de la red a un tamaño aleatorio entre el mínimo y el máximo configurados.
- `model.anchors`: lista con las relaciones de aspecto ancho-alto más frecuentes de los objetos a detectar. Estos anchors son usados por YOLO para proponer las regiones. Con el script, no mencionado anteriormente, `gen_anchors.py` se puede obtener fácilmente esta lista. Lo que hace el script es aplicar *k-means* sobre las etiquetas de las imágenes.
- `model.labels`: lista con los nombres de las clases de los objetos etiquetados en las imágenes.
- `model.data_load_method`: formato en el que están representadas las etiquetas de las imágenes. Acepta los valores `txt` y `voc` (XML).
- `train.train_image_folder`: ruta al directorio donde se encuentran las imágenes de entrenamiento.
- `train.train_annot`: ruta a un directorio (en el caso de anotaciones *voc*) o a un fichero (en el caso de anotaciones *txt*) con las anotaciones de las imágenes de entrenamiento.
- `train.cache_name`: ruta a un fichero que sirve de caché para las imágenes de entrenamiento. Esta caché, además de contener las ubicaciones de las imágenes y las anotaciones de las mismas, también contiene las dimensiones de las imágenes. Estas dimensiones son necesarias cuando hay que hacer una redimensión de la imagen, para poder adaptar las regiones de las anotaciones en consonancia. Esta caché evita tener que volver a leer cada una de las imágenes para volver a obtener sus dimensiones.
- `valid.valid_image_folder`, `valid.valid_annot` y `valid.cache_name`: son propiedades análogas a `train.train_image_folder`, `train.train_annot` y `train.cache_name` respectivamente, pero para las imágenes de test.
- `train.saved_weights_name`: ruta donde se va a guardar el modelo entrenado. Después de cada época, si el modelo ha mejorado con respecto a la época anterior, será guardado en la ruta configurada, reemplazando al anterior.
- `train.pretrained_weights`: ruta con un modelo preentrenado. Esta propiedad permite hacer una transferencia de conocimiento, bien partiendo de un modelo entrenado para detectar otros objetos distintos, o bien para continuar con un entrenamiento anterior.

- **train.batch_size**: tamaño del bloque de imágenes después del cual se aplica *backpropagation*.
- **train.nb_epochs**: número de veces que el modelo es entrenado con todo el conjunto de entrenamiento.
- **train.learning_rate**: número que permite ajustar cómo de rápido aprende el modelo.
- **train.early_stopping_patience**: número de épocas que tienen que pasar sin que el modelo mejore para que se detenga el aprendizaje de manera automática, aunque no se haya llegado al número de épocas configuradas.
- **train.reduce_lr_on_plateau_patience**: número de épocas que tienen que pasar sin que el modelo mejore para que se reduzca el **train.learning_rate** de manera automática.
- **train.ignore_thresh**: umbral por debajo del cual una predicción es ignorada por lo representar suficientemente a ninguno de los objetos a detectar. Se utiliza la *IoU* como unidad de medida. Si el valor de *IoU* entre la región predicha y las regiones a detectar es inferior al límite configurado, la predicción es rechazada.
- **valid.duplicate_thresh**: umbral por encima del cual dos predicciones se consideran la misma. Se utiliza la *IoU* como unidad de medida. Si el valor de *IoU* entre varias de las regiones predichas para una misma imagen es superior al umbral configurado, se considerarán la misma predicción y prevalecerá la que tenga un valor más alto de *IoU* con respecto a la región a detectar. Esto es lo que se conoce como *Non-Maximun Suppression* [21]
- **train.train_times**: número de veces que se recorre el conjunto de entrenamiento por cada época. Esto es útil cuando el conjunto de entrenamiento contiene pocas imágenes.

Formatos para las etiquetas

Se soportan dos formatos para las etiquetas de las imágenes: **txt** y **voc**. El formato **txt** consiste en un fichero de texto con una línea por cada imagen con el siguiente aspecto:

```
/tmp/imagen1.jpg 100,100,100,40,1 200,200,75,30,1
/tmp/imagen2.jpg 300,300,250,100,1
```

Código 2: Ejemplo de fichero con las etiquetas en formato txt. La primera de las imágenes tiene 2 etiquetas, la primera está ubicada en la posición (100, 100) y tiene unas dimensiones de 100x40 píxeles. La segunda etiqueta está ubicada en la posición (200, 200) y tiene unas dimensiones de 75x30 píxeles. La segunda imagen tiene una única etiqueta ubicada en (300, 300) con dimensiones 250x100. Todas las etiquetas son de la clase 1

En el formato voc existe un fichero *xml* para cada una de las imágenes con el siguiente aspecto:

```
<annotation>
  <folder>/tmp</folder>
  <filename>imagen1.jpg</filename>
  <size>
    <width>3680</width>
    <height>2760</height>
    <depth>3</depth>
  </size>
  <object>
    <name>pothole </name>
    <difficult>0</difficult>
    <bndbox>
      <xmin>100</xmin>
      <ymin>100</ymin>
      <xmax>200</xmax>
      <ymax>140</ymax>
    </bndbox>
  </object>
</annotation>
```

Código 3: Ejemplo de fichero con las etiquetas en formato voc

Generación de *anchors*

Como se ha comentado anteriormente, en la descripción de la configuración, existe el script `gen_anchors.py` que permite obtener un número determinado de anchors para un conjunto de imágenes. Para poder obtener este listado hace falta:

- Disponer de un conjunto de imágenes
- Disponer de las etiquetas de las imágenes en formato `txt` (el formato `voc` no está soportado en este caso)
- Haber creado un fichero de configuración. En esta configuración el atributo `model.anchors` es irrelevante, porque es lo que se va a generar

Con todo lo anterior se dispone de lo necesario para ejecutar el script `gen_anchors.py`:

```
> python gen_anchors.py --config config.json --anchors 9
```

Código 4: Cómo generar los *anchors*

Como resultado se obtendrá una lista con tantos anchors como se hayan indicado. Cada anchor está compuesto por una pareja de números, que representan el ancho y el alto del anchor.

Entrenamiento

Para poder entrenar un modelo hace falta lo siguiente:

- Disponer de un conjunto de imágenes de entrenamiento
- Disponer de las etiquetas de las imágenes en uno de los formatos soportados
- Haber generado los *anchors* para las imágenes de entrenamiento
- Haber creado un fichero de configuración

Una vez que se dispone de todo lo anterior, lo único que queda por hacer es ejecutar el script `train.py`:

```
> python train.py --config config.json
```

Código 5: Cómo lanzar el entrenamiento

Una vez finaliza el entrenamiento, y si se ha configurado el bloque `valid` en el fichero de configuración, se realizará una evaluación del modelo obtenido.

Evaluación

Para poder evaluar un modelo entrenado hará falta:

- Disponer de un conjunto de imágenes de test
- Disponer de las etiquetas de las imágenes en uno de los formatos soportados
- Disponer de un modelo ya entrenado
- Haber creado un fichero de configuración

Una vez se dispone de todo lo anterior, lo único que habrá que hacer es ejecutar el script `evaluate.py`:

```
> python evaluate.py --config config.json
```

Código 6: Cómo evaluar un modelo entrenado

La evaluación hace el cálculo de la *AP* tanto de una forma global como para cada una de las clases que se hayan configurado.

Predicción

Para realizar una predicción con el modelo ya entrenado es necesario:

- Disponer de una imagen o directorio con imágenes o vídeo
- Disponer de un modelo ya entrenado
- Haber creado un fichero de configuración

Con todo lo anterior, se ejecuta el script `predict.py`:

```

# para una imagen
> python predict.py -c config.json -i /tmp/imagen.jpg -o /tmp/pred/

# para un video
> python predict.py -c config.json -i /tmp/video.mp4 -o /tmp/pred/

# para un conjunto de imagenes
> python predict.py -c config.json -i /tmp/imagenes -o /tmp/pred/

```

Código 7: Cómo hacer una predicción con un modelo entrenado

En el directorio de salida habrá el mismo contenido que en la entrada modificado con las etiquetas que se han encontrado.

Transformación del modelo

Para explotar un modelo entrenado en un dispositivo móvil es necesario realizar una transformación del mismo. Para realizar esta transformación, *TFLite* que proporciona una serie de utilidades. En el siguiente bloque de código se muestra un ejemplo:

```

import tensorflow as tf

tflite_converter = tf.lite.TFLiteConverter
    .from_keras_model_file(<KERAS_H5_MODEL_PATH>)

tflite_model = tflite_converter.convert()

with open(<KERAS_TFLITE_DEST_PATH>, 'wb') as tflite_model_file:
    tflite_model_file.write(tflite_model)

```

Código 8: Cómo transformar un modelo entrenado con *Keras* a formato *TFLite*

Explotación del modelo

TFLite además de proporcionar herramientas para transformar modelos también proporciona librerías para explotarlos de diversas formas. En concreto, proporciona una librería java, que entre otras cosas, permite explotar un modelo en un dispositivo móvil *Android*.

En este primer bloque de código se muestra un ejemplo de cómo se puede cargar un modelo que forma parte de los *assets* de la aplicación móvil:

```

AssetFileDescriptor fileDescriptor = assets
    .openFd(<MODEL_TFLITE_FILENAME>);
FileInputStream inputStream = new FileInputStream(fileDescriptor
    .getFileDescriptor());
FileChannel fileChannel = inputStream.getChannel();
long startOffset = fileDescriptor.getStartOffset();
long declaredLength = fileDescriptor.getDeclaredLength();
MappedByteBuffer model = fileChannel.map(
    FileChannel.MapMode.READ_ONLY,
    startOffset, declaredLength);

```

```

Interpreter.Options interpreterOptions = new Interpreter.Options()
    .setNumThreads(1);
Interpreter tfLite = new Interpreter(model, interpreterOptions);

```

Código 9: Cómo cargar un modelo

En la variable `tfLite` se tiene disponible el modelo ya cargado, listo para ser explotado. Para ello, hay que proporcionarle los vectores que se espera como entrada e indicarle los vectores donde devolver el resultado. En el siguiente ejemplo se crea un único vector de entrada de dimensiones $1 \times 256 \times 256 \times 3$, que se correspondería con una imagen a color de 256×256 . Y se crean 2 vectores de salida de tamaños $8 \times 8 \times 18$ y $16 \times 16 \times 18$, que se corresponden con los vectores de salida de la versión YOLO *v3 tiny* con un tamaño de 256:

```

float[][][] floatValues = new float[1][256][256][3];

for (int i = 0; i < 256; i++) {
    for (int j = 0; j < 256; j++) {
        for (int k = 0; k < 3; k++) {
            floatValues[0][i][j][k] = 1.0f;
        }
    }
}

Object[] inputArray = {floatValues};

output1 = new float[1][8][8][18];
output2 = new float[1][16][16][18];

Map<Integer, Object> outputMap = new HashMap<>();
outputMap.put(0, output1);
outputMap.put(1, output2);

```

Código 10: Creación de vectores de entrada y de salida del modelo

El modelo se ejecuta de la siguiente manera:

```

tfLite.runForMultipleInputsOutputs(inputArray, outputMap);

```

Código 11: Cómo ejecutar el modelo

Una vez termine de ejecutarse el método `runForMultipleInputsOutputs`, en la variable `outputMap` estará disponible la salida del modelo lista para ser interpretada.

6.2. Evaluación de las técnicas

Se han entrenado dos versiones de YOLO: la versión *v3* y la versión *v3 tiny*. La versión *v3 tiny* es la versión indicada para ser ejecutada en un dispositivo móvil. La versión *v3* se ha entrenado para compararla con la *tiny*.

Para cada una de estas versiones se han entrenado varios modelos con distintos tamaños de red, por dos motivos principalmente: por una cuestión de rendimiento a la hora de ejecutar el modelo en un dispositivo móvil y por analizar cómo varía la precisión del modelo cambiando el tamaño de la red.

Además se han utilizado distintos conjuntos de entrenamiento para entrenar todas las variantes del modelo. El primero de los conjuntos de entrenamiento se corresponde con el conjunto íntegro original (denominado *completo*). Los resultados obtenidos con este conjunto de entrenamiento obtuvieron unos valores bajos para la métrica *AP*, y tras analizar los motivos, se observó que había una gran cantidad de baches demasiado pequeños que podían ser los causantes malos resultados. Por este motivo, se han utilizado dos conjuntos de entrenamiento adicionales aplicando filtros sobre los baches. En el primero de estos conjuntos de entrenamiento adicionales se han filtrado los baches con tamaño superior a 75x30 píxeles (denominado *filtro 75x30*) y en el segundo se han filtrado los baches con tamaño superior a 100x40 píxeles (denominado *filtro 100x40*). Para cada uno de estos conjuntos de entrenamiento adicionales se ha creado también su correspondiente conjunto de evaluación aplicando el mismo filtro.

Con todos los modelos resultantes obtenidos se ha realizado una doble evaluación. Por un lado se han evaluado con los conjuntos de evaluación correspondientes para cada uno de los conjuntos de entrenamiento (resultados en la tabla 2). Por otro lado se han evaluado con un conjunto de imágenes generado (resultados en la tabla 3). Este conjunto de evaluación (denominado *propio*) se compone de unas 30 imágenes de 4032x3024 píxeles, con unos 60 baches en total, obtenido desde la acera (a diferencia del original que fue obtenido desde el coche) y compuesto por fotos realizadas en España (a diferencia del original que fueron realizadas en Sudáfrica).

Versión YOLO	Tamaño	Juego datos	Épocas	Mejor AP
V3	256	completo	43	0.0747
V3	256	filtro 100x40	93	0.3077
V3	256	filtro 75x30	88	0.2513
V3	416	completo	18	0.1467
V3	416	filtro 100x40	93	0.4161
V3	416	filtro 75x30	93	0.3611
V3	640	completo	13	0.0186
V3	640	filtro 100x40	63	0.5475
V3	640	filtro 75x30	53	0.4106
V3 Tiny	256	completo	144	0.0046
V3 Tiny	256	filtro 100x40	136	0.0510
V3 Tiny	256	filtro 75x30	153	0.0392
V3 Tiny	416	completo	153	0.0145
V3 Tiny	416	filtro 100x40	153	0.1307
V3 Tiny	416	filtro 75x30	146	0.0869

Tabla 2: Resultados obtenidos con los conjuntos de evaluación originales

Versión YOLO	Tamaño	Juego datos	Épocas	Mejor AP
V3	256	propio completo	43	0.0289
V3	256	propio filtro 100x40	93	0.1018
V3	256	propio filtro 75x30	88	0.0179
V3	416	propio completo	18	0.0354
V3	416	propio filtro 100x40	93	0.0089
V3	416	propio filtro 75x30	93	0.0294
V3	640	propio completo	13	0.0017
V3	640	propio filtro 100x40	63	0.0342
V3	640	propio filtro 75x30	53	0.0961
V3 Tiny	256	propio completo	144	0.0086
V3 Tiny	256	propio filtro 100x40	136	0.0232
V3 Tiny	256	propio filtro 75x30	153	0.0371
V3 Tiny	416	propio completo	153	0.0000
V3 Tiny	416	propio filtro 100x40	153	0.0000
V3 Tiny	416	propio filtro 75x30	146	0.0006

Tabla 3: Resultados obtenidos con el conjunto de evaluación propio

7. Resultados

7.1. Resultados del proyecto

A continuación se van a mostrar algunos ejemplos de las predicciones obtenidas con los modelos que mejores resultados han obtenido, que son los que han sido entrenados con el juego de datos de entrenamiento *filtro 100x40*.



(a) Baches a detectar

(b) YOLO v3 tamaño 256x256



(c) YOLO v3 tamaño 416x416

(d) YOLO v3 tamaño 640x640

Figura 9: Ejemplo de predicción con modelos YOLO v3 de distintos tamaños. Arriba a la izquierda, la imagen con los baches a detectar en azul y en amarillo los baches que fueron descartados por el filtro 100x40. En el resto de las imágenes se pueden ver las predicciones realizadas en rojo.

En la figura 9 se muestran las predicciones realizadas por los modelos *YOLO v3*. Se trata de una imagen en la que originalmente se han etiquetado 2 baches, uno de los cuales se ha descartado por ser demasiado pequeño. Se puede observar que existe un defecto en el etiquetado, ya que entre los dos baches etiquetados

existe un tercer bache sin etiquetar. Aún habiendo filtrado los baches pequeños se puede comprobar que el modelo es capaz de detectarlos (en los modelos de tamaño 416x416 y 640x640). También se puede observar que el modelo de tamaño 640x640 es capaz de detectar el bache sin etiquetar.

En la figura 10 se muestran las predicciones realizadas por los modelos *YOLO v3 tiny* para la misma imagen. Únicamente el modelo de tamaño 416x416 es capaz de detectar el bache, aunque lo hace de manera poco precisa ya que la región detectada es demasiado grande y abarca también al bache sin etiquetar.



Figura 10: Misma predicción que en la figura 9, pero en esta ocasión con modelos *YOLO v3 tiny*.

En la figura 11 se muestran más predicciones realizadas por los modelos *YOLO v3*. En esta ocasión se trata de una imagen en la que hay múltiples baches, de los cuales únicamente se han mantenido 2 y el resto se han descartado por tener un tamaño demasiado pequeño. En esta ocasión los 3 modelos detectan baches de forma correcta. El único modelo que detecta los baches esperados es el de tamaño 640x640. Además de detectar los baches detectados, es capaz de detectar uno de los baches que fue descartado por tamaño. Los otros dos modelos de tamaño inferior únicamente detectan uno de los baches esperados, aunque son capaces de detectar también algunos de los baches descartados.

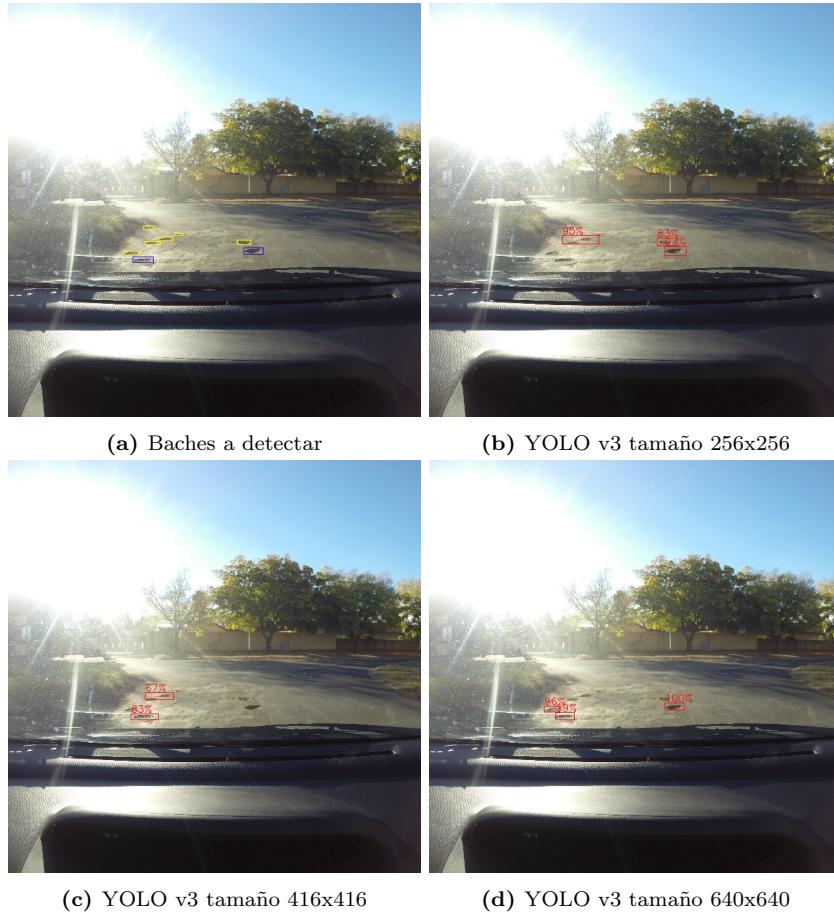


Figura 11: Ejemplo de predicción con modelos YOLO v3 de distintos tamaños. Arriba a la izquierda, la imagen con los baches a detectar en azul y en amarillo los baches que fueron descartados por el filtro 100x40. En el resto de las imágenes se pueden ver las predicciones realizadas en rojo.

En la figura 12 se muestran las predicciones realizadas por los modelos *YOLO v3 tiny* para el segundo ejemplo. En esta ocasión ambos modelos son capaces de detectar baches de forma correcta. Además el modelo de tamaño 416x416 es capaz de identificar los dos baches esperados.



(a) YOLO v3 tiny tamaño 256x256

(b) YOLO v3 tiny tamaño 416x416

Figura 12: Misma predicción que en la figura 11, pero en esta ocasión con modelos YOLO v3 tiny.

En la figura 13 se muestra otro ejemplo de predicciones realizadas por los modelos *YOLO v3*. En esta ocasión también se trata de una imagen en la que hay múltiples baches, de los cuales se ha mantenido un único bache, el resto han sido descartados descartado por tener un tamaño demasiado pequeño. En esta ocasión uno de los modelos, el de tamaño 416x416, es incapaz de detectar ningún bache. El modelo más pequeño, detecta uno de los baches que ha sido descartado, y la predicción es mejor al original ya que la región predicha se ajusta más al bache. El modelo más grande ha sido capaz de detectar el bache original y además detecta otro de los baches que fue descartado.



Figura 13: Ejemplo de predicción con modelos YOLO v3 de distintos tamaños. Arriba a la izquierda, la imagen con los baches a detectar en azul y en amarillo los baches que fueron descartados por el filtro 100x40. En el resto de las imágenes se pueden ver las predicciones realizadas en rojo.

En la figura 14 se pueden ver las predicciones realizadas por los modelos *YOLO v3 tiny* para el ejemplo anterior. Sólo uno de los modelos detecta uno de los baches.



(a) YOLO v3 tiny tamaño 256x256 (b) YOLO v3 tiny tamaño 416x416

Figura 14: Misma predicción que en la figura 13, pero en esta ocasión con modelos YOLO v3 tiny.

En la figura 15 se muestra un ejemplo más de predicciones realizadas por los modelos *YOLO v3*. En esta ocasión también se trata de una imagen en la que hay múltiples baches, de los cuales se han mantenido dos. Todos los modelos han sido capaces de detectar los baches originales, aunque no de la misma forma. Uno de los baches originales se encuentra contiguo a otro descartado, es por esto que las regiones propuestas por los dos modelos más pequeños para este bache sean más grandes, abarcando ambos baches.



Figura 15: Ejemplo de predicción con modelos YOLO v3 de distintos tamaños. Arriba a la izquierda, la imagen con los baches a detectar en azul y en amarillo los baches que fueron descartados por el filtro 100x40. En el resto de las imágenes se pueden ver las predicciones realizadas en rojo.

En la figura 16 se pueden ver las predicciones realizadas por los modelos *YOLO v3 tiny* para el ejemplo anterior. Ambos modelos detectan los baches, aunque se están haciendo predicciones extrañas que son erróneas.



(a) YOLO v3 tiny tamaño 256x256

(b) YOLO v3 tiny tamaño 416x416

Figura 16: Misma predicción que en la figura 15, pero en esta ocasión con modelos YOLO v3 tiny.

En la figura 17 se muestra un ejemplo más de predicciones realizadas por los modelos *YOLO v3*. En esta ocasión también se trata de una imagen en la que hay múltiples baches, de los cuales se mantiene uno. Este ejemplo es un poco diferente a los anteriores porque este no es un caso evidente para el ojo humano, aún así, todos los modelos han sido capaces de detectar el bache original.



Figura 17: Ejemplo de predicción con modelos YOLO v3 de distintos tamaños. Arriba a la izquierda, la imagen con los baches a detectar en azul y en amarillo los baches que fueron descartados por el filtro 100x40. En el resto de las imágenes se pueden ver las predicciones realizadas en rojo.

En la figura 18 se pueden ver las predicciones realizadas por los modelos *YOLO v3 tiny* para el ejemplo anterior. Sólo uno de los modelos es capaz de hacer una predicción, y además errónea.



(a) YOLO v3 tiny tamaño 256x256

(b) YOLO v3 tiny tamaño 416x416

Figura 18: Misma predicción que en la figura 17, pero en esta ocasión con modelos YOLO v3 tiny.

En la figura 19 se muestra un último ejemplo de predicciones realizadas por los modelos *YOLO v3*. En esta ocasión también se trata de una imagen en la que hay un único bache. En esta ocasión todos los modelos tienen un comportamiento errático. Además de detectar el bache, se han visto afectados por las hojas en el asfalto realizando predicciones erróneas.



(a) Baches a detectar

(b) YOLO v3 tamaño 256x256



(c) YOLO v3 tamaño 416x416

(d) YOLO v3 tamaño 640x640

Figura 19: Ejemplo de predicción con modelos YOLO v3 de distintos tamaños. Arriba a la izquierda, la imagen con los baches a detectar en azul y en amarillo los baches que fueron descartados por el filtro 100x40. En el resto de las imágenes se pueden ver las predicciones realizadas en rojo.

En la figura 20 se pueden ver las predicciones realizadas por los modelos *YOLO v3 tiny* para el ejemplo anterior. Estos modelos se han visto afectados de la misma manera por las hojas en el asfalto.



(a) YOLO v3 tiny tamaño 256x256

(b) YOLO v3 tiny tamaño 416x416

Figura 20: Misma predicción que en la figura 19, pero en esta ocasión con modelos YOLO v3 tiny.

8. Conclusiones

8.1. Evaluación del proyecto

La evaluación del proyecto en general es muy positiva, habiéndose cumplido sobradamente tanto con todos los objetivos iniciales como con los requisitos analizados.

Teniendo en cuenta los objetivos iniciales del proyecto y los requisito del mismo, lo más indicado habría sido utilizar únicamente una versión *tiny* de YOLO, ya que fueron concebidas para ser explotados en dispositivos con pocos recursos. Se ha decidido implementar y entrenar modelos *YOLO v3* para hacer una comparativa con su versión *tiny*.

Todo el proyecto ha sido desarrollado en la plataforma *Google Colab*, la cual presenta una serie de limitaciones en cuando a la capacidad del hardware disponible y en cuanto al tiempo de uso del mismo. Las limitaciones de capacidad del hardware han impactado al entrenamiento, teniendo que limitar tanto el tamaño de la red como el *batch size* de la red. Las limitaciones de tiempo de uso han forzado realizar en varias fases el entrenamiento de los modelos, sobre todo los más grandes. En cada fase se ha realizado una *transferencia de conocimiento*, teniendo como punto de partida el modelo resultante de la fase anterior, salvo en la fase inicial que se tiene como punto de partida el modelo entrenado publicado por los autores de *YOLO*. Tras finalizar cada fase se ha realizado una evaluación del modelo obtenido y se ha comparado con el de la fase anterior para decidir si se prosigue con el entrenamiento.

Los modelos han sido entrenados pocas épocas debido a la cantidad de tiempo que requería el entrenamiento y por la limitación de tiempo de uso del hardware. La plataforma utilizada para el desarrollo del proyecto no está pensada para hacer un uso muy intensivo de la misma. Si es el caso y se lleva la infraestructura a sus límites, eres penalizado y las sucesivas solicitudes de hardware son rechazadas. Para el entrenamiento de varios de los modelos se ha llevado la plataforma al límite en repetidas ocasiones, haciendo uso de toda la capacidad de GPU (12 GB) y de todo el tiempo de uso disponible (12 horas). Así como en la fase inicial de entrenamiento era sencillo conseguir la asignación de recursos, en la fase final ha sido lo contrario, resultando difícil la asignación de recursos, teniendo que solicitarlos repetidas ocaciones durante varias horas hasta conseguirlos finalmente.

Al transformar los modelos a TFLite y evaluarlos desde python y java, los resultados obtenidos no son exactamente iguales. Existen pequeñas diferencias en las salidas del modelo a partir del 4-6 decimal. No se ha encontrado una explicación para esto, aunque estas diferencias no alteran demasiado las predicciones.

El tiempo de la inferencia en el dispositivo móvil ha resultado superior al esperado, obteniendo unos valores bajos de *FPS* para las imágenes procesadas. Con un móvil de gama media (Nokia 7 plus) se alcanzan los 5-7 FPS. Con un móvil

de gama alta (OnePlus 7 pro) se alcanzan unos 12 FPS.

8.2. Alternativas y posibles mejoras que podrían haberse aplicado al proyecto (trabajos futuros)

El principal punto de mejora se encuentra en los datos utilizados para el proyecto. Por un lado es un conjunto de datos un poco escaso y el etiquetado del mismo podría mejorarse. Con respecto al etiquetado, lo que se ha observado es que existen imágenes en las cuales hay baches que no han sido etiquetados. Para estos algoritmos es importante etiquetar todos los objetos a detectar existentes en la imagen.

Tras ver las diferencias de los resultados obtenidos con las imágenes originales y con las imágenes propias, se han observado notables diferencias en las características de los baches. En las imágenes originales los baches presentan piedras, arena, hojas. Sin embargo en los baches propios hay más asfalto, debido a que las carreteras son reasfaltadas, y los baches dejan al descubierto el asfalto original (ejemplo en figura 21). Por lo que sería interesante disponer de un juego de datos obtenido en España para realizar el entrenamiento y ver cómo varían los resultados.



Figura 21: A la izquierda un ejemplo de bache típico presente en las imágenes originales. A la derecha un ejemplo de bache típico de las imágenes de España

Uno de los factores más determinantes para la precisión obtenida por los modelos ha sido el tamaño de los baches con respecto al de la imagen. Sería interesante disponer de un juego de datos en el que las fotos hayan sido tomadas desde el exterior del vehículo y con un encuadre de tal forma que los baches, aun siendo pequeños, tengan un tamaño representativo en la imagen. Esto podría conseguirse encuadrando la zona del asfalto frente al vehículo sin que aparezca paisaje al fondo ni en los laterales.

Otra vía de mejora a explorar para aumentar el número de *FPS* procesadas por el modelo en el dispositivo móvil consistiría en realizar una transformación sobre el modelo para obtener uno *quantized* [22].

Habrá que estar al tanto del soporte de GPU en *TFLite* ya que actualmente se encuentra en fase experimental. Hoy en día es bastante frecuente que los dispositivos móviles tengan incorporada una GPU, por lo que sería muy interesante poder hacer uso de la misma. En la aplicación móvil se ha hecho uso de la misma, aunque no se ha obtenido una mejora, debido precisamente a que se encuentra en fase experimental y a que una de las operaciones del modelo (*BatchNormalization*) [23] no se encuentra soportada por el momento. Bien podría esperarse a que se soporte esta operación o bien se podría estudiar hacer una variación del modelo para remplazar esta operación por otra soportada sin que se afecte a la precisión del mismo.

8.3. Conclusiones personales

El proyecto me ha resultado un reto desafiante y gratificante ya que me ha permitido ampliar mis conocimientos haciendo un simulacro de lo que podría ser un proyecto profesional.

Tener un juego de datos con abundantes imágenes y correctamente etiquetadas es clave para obtener buenos resultados. Pero esto no es suficiente, ya que al igual que ocurre con otras técnicas es primordial conocer los datos. Gracias al estudio del aspecto y tamaño de los baches he podido adaptar el preprocesamiento de las imágenes y crea distintos subconjuntos de entrenamiento que han mejorado los resultados del modelo.

Si la detección de objetos es un problema complejo a resolver, creo que la detección de objetos pequeños incrementa bastante la complejidad en todos los aspectos: etiquetado de imágenes, entrenamiento y evaluación del modelo.

Aunque el número de *FPS* alcanzados en el dispositivo móvil es inferior al esperado, he conseguido alcanzar una cifra que permite una fluidez aceptable. Como se ha comentado en el punto anterior, existen múltiples vías de mejora de este apartado que creo que conseguirían mejorar este aspecto y hacer el producto válido para un uso productivo.

Referencias

- [1] Ayuntamiento de Madrid. *Sistema de Sugerencias y Reclamaciones*. URL: <https://www.madrid.es/portales/munimadrid/es/Inicio/El-Ayuntamiento/Contacto/Sugerencias-y-reclamaciones?vgnextfmt=default&vgnextchannel=5eadc1ab4fd86210VgnVCM2000000c205a0aRCRD>. (accedido: 01/09/2019).
- [2] Ayuntamiento de Madrid. *Informe de sugerencias y reclamaciones del ayuntamiento de Madrid de 2018*. URL: https://www.madrid.es/UnidadesDescentralizadas/UGDefensorContribuyente/05_Publicaciones/informes/INFORME_1%20semestre_2018_SyRversion_definitiva_18_diciembre.pdf.
- [3] Javier Rey. *Object Detection with Deep Learning: The Definitive Guide*. URL: <https://tryolabs.com/blog/2017/08/30/object-detection-an-overview-in-the-age-of-deep-learning>. (accedido: 29/08/2019).
- [4] Joyce Xu. *Deep Learning for Object Detection: A Comprehensive Review*. URL: <https://towardsdatascience.com/deep-learning-for-object-detection-a-comprehensive-review-73930816d8d9>. (accedido: 03/09/2019).
- [5] Arthur Ouaknine. *Review of Deep Learning Algorithms for Object Detection*. URL: <https://medium.com/zylapp/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852>. (accedido: 03/09/2019).
- [6] Ankit Sachan. *Zero to Hero: Guide to Object Detection using Deep Learning: Faster R-CNN, YOLO, SSD*. URL: [https://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/amp/](https://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/). (accedido: 03/09/2019).
- [7] Dhruv Parthasarathy. *A Brief History of CNNs in Image Segmentation: From R-CNN to Mask R-CNN*. URL: <https://blog.athelas.com/a-brief-history-of-cnns-in-image-segmentation-from-r-cnn-to-mask-r-cnn-34ea83205de4>. (accedido: 03/09/2019).
- [8] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2014. URL: <https://arxiv.org/pdf/1311.2524.pdf>.
- [9] J.R.R. Uijlings et al. *Selective Search for Object Recognition*. 2012. URL: <http://www.huppelen.nl/publications/selectiveSearchDraft.pdf>.
- [10] Ross Girshick. *Fast R-CNN*. 2015. URL: <https://arxiv.org/pdf/1504.08083.pdf>.
- [11] Shaoqing Ren. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2016. URL: <https://arxiv.org/pdf/1506.01497.pdf>.
- [12] Joseph Redmon. *YOLOv3: An Incremental Improvement*. 2018. URL: <https://arxiv.org/pdf/1804.02767.pdf>.
- [13] experiencor. URL: <https://github.com/experiencor/keras-yolo3>.

- [14] HoracceFeng. URL: <https://github.com/HoracceFeng/keras-yolo3-tiny>.
- [15] dicastro. URL: <https://github.com/dicastro/tfm>.
- [16] Felipe Muller. *Nienaber Potholes 2 Complex*. URL: <https://www.kaggle.com/felipemuller5/nienaber-potholes-2-complex>. (accedido: 26/08/2019).
- [17] Juan Ignacio Bagnato. *¿Cómo funcionan las Convolutional Neural Networks? Visión por Ordenador*. URL: <https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/>. (accedido: 06/09/2019).
- [18] Wikipedia. *Convolutional neural network*. URL: https://en.wikipedia.org/wiki/Convolutional_neural_network. (accedido: 06/09/2019).
- [19] Wikipedia. *Redes neuronales convolucionales*. URL: https://es.wikipedia.org/wiki/Redes_neuronales_convolucionales. (accedido: 06/09/2019).
- [20] Rising Odeguia. *Transfer learning and Image classification using Keras on Kaggle kernels*. URL: <https://towardsdatascience.com/transfer-learning-and-image-classification-using-keras-on-kaggle-kernels-c76d3b030649>. (accedido: 06/09/2019).
- [21] Rasmus Rothe. *Non-Maximum Suppression for Object Detection by Passing Messages between Windows*. 2015. URL: https://www.vision.ee.ethz.ch/publications/papers/proceedings/eth_biwi_01126.pdf.
- [22] Tensorflow. *Post-training quantization*. URL: https://www.tensorflow.org/lite/performance/post_training_quantization. (accedido: 07/09/2019).
- [23] F D. *Batch normalization in Neural Networks*. URL: <https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c>. (accedido: 07/09/2019).
- [24] Joseph Redmon. *You Only Look Once: Unified, Real-Time Object Detection*. 2015. URL: <https://arxiv.org/pdf/1506.02640.pdf>.
- [25] Jonathan Hui. *mAP (mean Average Precision) for Object Detection*. URL: https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173. (accedido: 29/08/2019).