Universidad Rey Juan Carlos

Práctica Obligatoria 1 - Visión Artificial
Detección de Objetos

Alejandro Herrera Cañas
Jorge Guerrero Ramos
(*Grupo I*)

20 de abril de 2020

1. Introducción

En este documento se detalla el trabajo realizado para implementar los sistemas de reconocimiento de objetos (concretamente coches) propuestos en el enunciado así como una serie de estadísticas sobre el impacto que estos han tenido y unas conclusiones.

2. Implementación de los sistemas

En este apartado se describen los pasos seguidos para la implementación de los diferentes sistemas de detección de vehículos:

2.1. Detección de coches mediante puntos de interés (`deteccion_orb.py`)

Para que el sistema funcione primero se requiere entrenarlo con las imágenes de training proporcionadas. Para ello, el método `entrenamiento_orb` procesa las imágenes, las convierte a escala de grises utilizando `cv2.cvtColor` y extrae sus puntos de interés y descriptores con `cv2.detectAndCompute`:

- Por un lado, los keypoints se almacenan en una lista común (`kps_training`) para utilizarlos más adelante.
- Los descriptores se guardan en una estructura del tipo `cv2.FlannBasedMatcher` de forma que estén correctamente indexados.

Además, en el fichero *Python* donde se desarrolla este sistema, se encuentra el método encargado de leer las correspondientes imágenes desde una carpeta (`carga_imagenes_carpeta`).
En `procesamiento_img_orb` se han implementado los pasos necesarios para extraer los descriptores y los puntos de interés para cada imagen de test, de manera que se puedan conocer los *matches* que existen con los ya extraídos previamente en el entrenamiento y a partir de ahí quedarse con el mejor (el que aparentemente contiene el frontal del coche). Recibe como parámetros la imagen con la que se trabaja, así como el tamaño de la imagen de entrenamiento ya que se le realiza un reescalado previo. El funcionamiento del método se describe a continuación:

1. Se crea el detector orb con `cv2.ORB_create`. Los parámetros son los definidos en el enunciado de la práctica.
2. Como se ha mencionado una decisión de implementación que se ha decidido es la de redimensionar (`cv2.resize`) la imagen de test al tamaño de las imágenes de entrenamiento (ya que tienen el mismo tamaño). De esta manera no hay una imagen más grande que otra.

3. También adicionalmente se potencian los detalles de la imagen usando `cv2.detailEnhance`.

4. Con dicha imagen se extraen tanto sus descriptores como sus puntos de interés con los que se forma un iterador que es el que nos interesa para poder realizar el procesamiento.

5. Se declara la matriz de votación, que es una matriz de ceros cuyo tamaño es el de la imagen y que tiene como cometido ir acumulando los votos que se emitan.

6. Se crea el vector de votación Hough llamando al método `votacion_hough`, al que se le pasa como parámetros la imagen junto a los puntos de interés de cada conjunto. A continuación se detalla el funcionamiento de dicho método:

    6.1. Se crea un vector que parte de la posición del punto de interés (`pt`) hasta el punto central de la imagen que se está procesando (dividiendo cada una de sus coordenadas entre dos), teniendo en cuenta el tamaño (`size`) tanto de los *keypoints* de entrenamiento como de *test* y cuyo ángulo está definido en relación al ángulo (`angle`) de la imagen de test.

    6.2. Se obtiene el módulo de dicho vector.

    6.3. Por último el vector se escala teniendo en cuenta el tamaño del keypoint la imagen de test (en nuestro caso se divide entre 10 cada coordenada). Es el que se retorna.

7. Este vector de votación *Hough* se construye para cada cada uno de los *matches* potenciales entre el descriptor de la imagen de *test* y la *query* de descriptores procesados de *training* (indexados previamente en un *FlannBasedMatcher*), así como sus 5 mejores vecinos para cada uno de ellos (ya que se ha establecido k=5 en `knnMatch`).

8. El criterio para realizar la votación es que mientras que las coordenadas del vector no se salgan del tamaño de la imagen se considerará como voto válido. Si esto se cumple se sumará 1 a la posición del vector de votación que se corresponde con las coordenadas.

9. Una vez que se completa la matriz de votación hay que extraer el *mejor*, es decir, el que haya recibido más votos. Será la posición de la matriz de votación con el número más alto (aplicando `argmax()` sobre dicha matriz). Como al inicio se construyó un iterador *keypoint*-descriptor, es necesario deshacerlo para obtener únicamente las coordenadas del punto. Esto se consigue gracias al método `unravel_index()`

10. Teniendo las coordenadas del mejor punto, se dibuja un círculo sobre el mismo utilizando `cv2.circle` con (¡Ojo! Hay que volver a escalar las coordenadas).

11. Para acabar, el método `detector_coches_orb` tiene como cometido aplicar el procesamiento (utilizando los métodos explicados anteriormente) al grupo de imágenes de test que se le pasan por parámetros y mostrar la salida (las imágenes ya procesadas).

Las figuras 1a y 1b muestran cómo el sistema detecta satisfactoriamente el frontal del coche:

(a) Toma como referencia la insignia del coche

(b) La insignia del coche no se encuentra en posición frontal por lo que toma como referencia la matrícula

Figura 1: Frontales de ambos coches detectados perfectamente

Sin embargo, también se han encontrado particularidades en la ejecución del detector ORB sobre las imágenes de test. Éstas son:

- En las dos primeras imágenes por ejemplo del conjunto de test que lee, detecta el espejo derecho del coche como se observa en las figuras 2a y 2b.
- Cuando en la imagen aparecen dos coches el sistema se confunde, como sucede en la Figura 3.

(a)

(b)

Figura 2: El detector ORB se fija en el espejo derecho de los coches

Figura 3: El detector parece quedarse en el punto medio de los frontales de ambos coches

2.2. Detección de coches usando cv2.CascadeClassifier (`deteccion_haar.py`)

Los parámetros de `cv2.detectMultiScale` se han fijado en base a hacer diferentes pruebas variando el valor de los mismos y se ha llegado la conclusión de que el sistema funciona satisfactoriamente usando un factor de escala muy pequeño (cercano a 1), incluyendo los 5 vecinos más cercanos al rectángulo candidato (*minNeighbours*), así como un tamaño mínimo de rectángulo (*minSize*) de 50x50 ya que el frontal de un coche se considera un objeto relativamente grande.
Una vez establecidas estas condiciones para el detector de objetos, se procede a pasar por cada imagen.
El método `cv2.detectMultiScale` devuelve las coordenadas de uno o más rectángulos donde el método *cree* que puede haber un frontal de coche. Antes de poder volver a mostrar la imagen procesada es necesario dibujar sobre la misma el rectángulo en sí, usando las coordenadas anteriormente mencionadas y el método `cv2.rectangle`.
Las figuras 4, 5 y 6 muestran cómo el sistema es capaz de detectar el frontal de los coches de forma adecuada (incluso cuando éstas tienen baja resolución):

Figura 4: La imagen tiene baja resolución

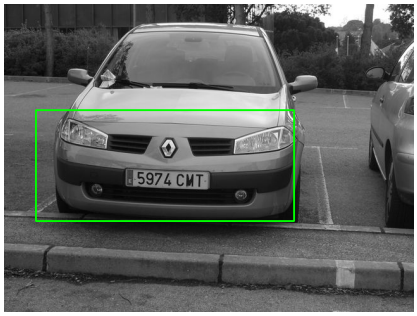


Figura 5: Imagen con buena resolución

Figura 6: La posición del coche no es totalmente frontal

Algunas particularidades del sistema son:

- No se reconoce el frontal del coche cuando este tiene líneas muy agresivas y/o esquinas (ver Figura 7).
- En algunas imágenes, como en la Figura 8, el sistema no detecta nada.
- Cuando en una misma escena se encuentra con varios coches en ocasiones es capaz de detectarlos como se muestra en la Figura 9.

Figura 7: El detector funciona de manera errónea

Figura 8: No se detecta ningún objeto

Figura 9: Coches en una misma imagen detectados simultáneamente

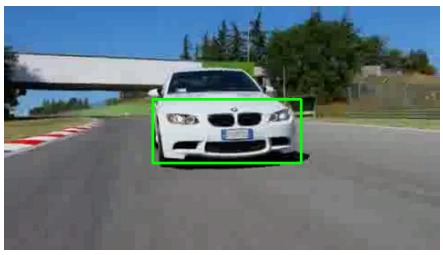
2.3. Detección del coche en secuencias de vídeo (`deteccion_video.py`)

Como indica el enunciado de la práctica, el objetivo de este apartado era aplicar las funciones ya definidas anteriormente a las secuencias de vídeo proporcionadas. Al entender un vídeo como una sucesión de imágenes (*frames*), lo único que habría que hacer es extraer los mismos y procesarlos con los métodos ya definidos con normalidad. Para lograr esto se usa el método `cv2.videoCapture` para

capturar el archivo de vídeo y un bucle *for* que procese cada *frame* extraído con `read()` sobre la captura de vídeo.
Se ha conseguido que el sistema cumpla satisfactoriamente los requisitos de diseño usando el clasificador en cascada. Sin embargo el detector ORB para los vídeos tiene dificultades para seguir la trayectoria observable de los coches. En la Figura 10 se muestra cómo funciona el detector:

Figura 10: Cuando el frame es lo suficientemente estable el detector funciona, aunque solo sobre un coche

Las figuras 11a y 11b muestran la ejecución del sistema de reconocimiento de coches usando `cv2.cascadeClassifier`:



(a) El caso básico, un solo coche en la escena

(b) El sistema detecta a ambos coches en el frame

Figura 11: Detección haar en secuencias de vídeo

De manera complementaria se ha decidido añadir la opción de poder guardar dichas secuencias procesadas en formato de vídeo, usando `cv2.VideoWriter`. Al considerarse una característica opcional se ha comentado el código que posibilita tal función.

3. Estadísticas

3.1. Tiempos de ejecución

Como medida razonable se ha tenido en cuenta el **tiempo de procesamiento medio por imagen** del sistema desarrollado (estos tiempos dependen en cierta medida de las especificaciones del sistema en el que se ejecuta, por lo que pueden variar). Tampoco se considera el tiempo de carga de las imágenes así como del entrenamiento ORB. Por tanto:

- Para el detector ORB: **0,5632** segundos.
- Para el detector haar: **0,0974** segundos.

En el caso del sistema de reconocimiento de coches en secuencias de vídeo, se ha decidido medir el tiempo total de procesamiento de los mismos. Queda de la siguiente manera:

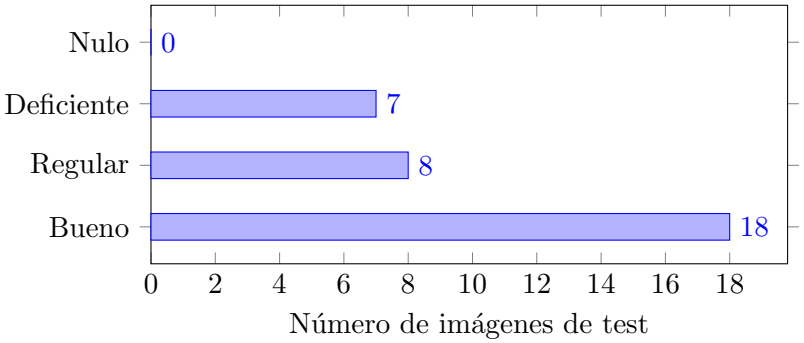
- En la secuencia de vídeo *video1.wmv*:
  - Usando el detector ORB: **1768,5195** segundos
  - Usando el detector haar: **56,3392** segundos.

- En la secuencia de vídeo *video2.wmv*:
  - Usando el detector ORB: **304,9955** segundos.
  - Usando el detector haar: **7,8398** segundos.

3.2. Calidad del sistema

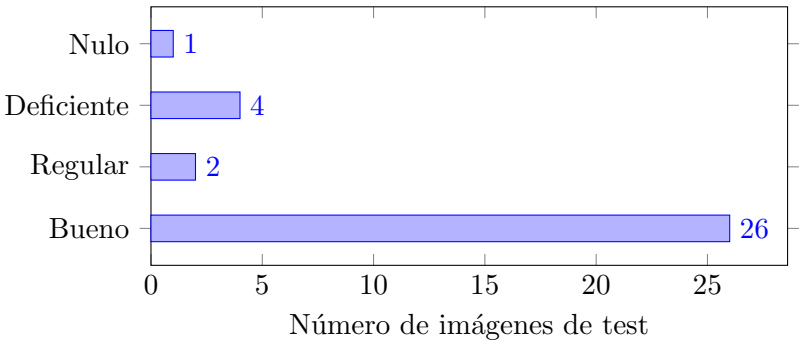
Para el detector basado en puntos de interés se ha establecido la siguiente métrica para medir su calidad:

- Es **bueno** cuando el círculo se encuentra sobre la insignia del coche o en sus alrededores.
- Es **regular** cuando estando en el frontal se encuentra bastante alejado del centro aproximado.
- Es **deficiente** cuando detecta algún otro elemento del coche que no tiene que ver con el frontal (o detecta otro coche de la escena).
- Es **nulo** cuando no detecta nada.



Continuando, para el detector `cascadeClassifier`, la forma de medir su calidad que se ha establecido es la que se muestra a continuación. Teniendo en cuenta la imagen de salida:

- Es **bueno** cuando el rectángulo encuadra el frontal del coche sin ningún tipo de dudas.
- Es **regular** cuando se podría decir que detecta el frontal del coche pero con imprecisiones.
- Es **deficiente** cuando detecta algún objeto del frontal, pero no el frontal en sí.
- Es **nulo** cuando no detecta nada



Se ha descartado la idea de aplicar la misma métrica sobre los sistemas en secuencias de vídeo debido a su complejidad.

4. Conclusiones

Tras la realización de esta práctica se han llegado a una serie de conclusiones:

- Elaborar un sistema de reconocimiento de objetos no supone una gran complejidad en sí mismo; realmente la complejidad reside en conseguir que ese sistema sea lo más eficaz posible.
- Hay que prestar atención cuando se trabaja con un gran fichero de imágenes o con un vídeo grande, ya que pueden surgir problemas de memoria si se intentan almacenar todos.
- La calidad de las imágenes de entrenamiento influye en gran medida en el funcionamiento final del sistema.