

Práctica 2. Extending a multi-object tracker

Pablo Aguilera Onieva y Fernando Fernández del Cerro

I. INTRODUCCIÓN

Este proyecto trata sobre el estudio y mejora de un modelo de aprendizaje profundo para seguimiento de múltiples objetos. Se parte de un modelo base, constituido por un *detector de objetos* de arquitectura *Fast-RCNN*, entrenado en un conjunto de datos etiquetados **MOT16**, y un seguidor de objetos (*tracker*) que asigna detecciones actuales a detecciones anteriores mediante Intersección sobre Unión de forma codiciosa (*greedy*). Así, se mejora la puntuación *MOTA* lograda optimizando ciertos aspectos del modelo base.

En cuanto a las tareas obligatorias, que constarían de la realización y entendimiento del tutorial provisto para comprender el funcionamiento del modelo base, el estudio de los scripts `tracker.py` y `object-detector.py` y, por último, el diseño, implementación y evaluación de las mejoras.

Respecto a las mejoras que se han considerado hacer, se trata de tener solo en cuenta detecciones de alta calidad del detector, seleccionando (*tuning*) el umbral óptimo que maximiza la puntuación F1-score en la detección de objetos, así como la aplicación del *algoritmo húngaro* [4] para la asociación de datos, combinación que conduce a una mejora muy significativa en las métricas utilizadas para la evaluación del modelo.

II. MÉTODO

El modelo base define un detector de objetos, el cual consiste en una clase `FRCNN_FPN`, heredada de `FasterRCNN` de la librería `torchvision`. Ésta está formada por los siguientes atributos:

- `GeneralizedRCNNTransform`: Módulo encargado de preprocessar las imágenes de entrada. Realiza operaciones de normalización y de cambio de dimensiones en los datos de entrada.
- `BackBoneWithFPN`: Arquitectura de red troncal para la extracción de las características de la imagen de entrada. Para este caso se usa una *CNN* seguida de una *FPN*.
- `IntermediateLayerGetter`: Módulo que extrae los mapas de características en diferentes capas permitiendo el acceso a ella.
- `Bottleneck`: Este módulo es responsable de reducir las dimensiones y aumentar la cantidad de canales en los mapas de características. Estos bloques constan de una serie de capas de convolución, con normalización en batches y con funciones de activación *ReLU*.
- `FeaturePyramidNetwork`: Este módulo tiene como objetivo principal abordar el desafío de detectar objetos a diferentes escalas en una imagen. Esta

pirámide consiste en múltiples mapas de características a diferentes escalas. Cada nivel de la pirámide representa una escala diferente de la imagen original, desde detalles finos hasta características de alto nivel.

- `RoIHeads`: Se encarga de procesar las regiones de interés propuestas generadas por el módulo de detección.
- `FastRCNNPredictor`: Módulo para predecir la clase y la regresión del cuadro delimitador de los *RoIs*.
- `Conv2d`: Capa de convolución de *Pytorch*.
- `LastLevelMaxPool`: Se aplica al último nivel de la pirámide de características para reducir su resolución espacial. Por lo general, se utiliza para igualar las resoluciones de los mapas de características de diferentes niveles de la pirámide.

La estructura del modelo sigue un enfoque escalonado, donde la red principal extrae las características de la imagen de entrada, mientras que la *Feature Pyramid Network (FPN)* crea múltiples mapas de características en distintas escalas. Estos mapas de características son esenciales para la identificación de regiones y la detección de objetos en las etapas posteriores del modelo *Faster R-CNN*.

Otro de los componentes principales del modelo es el seguidor de objetos, que se encarga de las asignaciones de las detecciones con los seguimientos, decidiendo mantener, crear o eliminar objetos seguidos frame a frame. El fichero `tracker.py` define las clases `Tracker`, encargada de rastrear objetos, y `Track`, que representa un seguimiento individual para un objeto detectado. La clase `Tracker` está compuesta de los siguientes métodos:

- `__init__(self, obj_detect)`: Inicializa el `tracker` con un módulo de detección de objetos.
- `reset(self, hard=True)`: Reinicia el `tracker` borrando los seguimientos, el número de seguimiento, los resultados y el índice de imágenes.
- `add(self, new_boxes, new_scores)`: Inicializa nuevos objetos de seguimiento con los cuadros delimitadores y puntuaciones proporcionadas y los guarda.
- `get_pos(self)`: Devuelve las posiciones de todos los seguimientos activos.
- `data_association(self, boxes, score)`: Asocia los datos con los seguimientos existentes y las cajas recién detectadas según el criterio *IoU*.
- `step(self, frame)`: Seguimiento de un único fotograma mediante la detección de objetos, la asociación y la actualización de los resultados.
- `get_results(self)`: Devolver los resultados del seguimiento

La clase `Track` está compuesta de los siguientes atribu-

tos:

- `id`: ID de seguimiento.
- `box`: Cuadro delimitador del objeto detectado.
- `score`: Confianza del detector sobre el objeto detectado.

En el archivo donde se hace la llamada al modelo y ejecuta el resto de ficheros, se genera la clase `TrackerIoUAssignment` que es heredada de la clase `tracker` y sobreajusta la función `data_association` para añadirle el criterio *IoU* que determina las mejores coincidencias.

Otro fichero importante es `object_detector.py`. Esta clase encapsula un modelo *Faster R-CNN* con una *Feature Pyramid Network* (FPN) como columna vertebral, proporcionando una interfaz para inicializar el modelo con un número específico de clases y realizar detecciones en imágenes de entrada. La función `detect` permite a los usuarios realizar detecciones utilizando este modelo y obtener las cajas delimitadoras y las puntuaciones de las detecciones resultantes. Como *backbone* utiliza una arquitectura *ResNet-50* sin utilizar los pesos pre-entrenados y el parámetro `nms_thresh` indica el umbral que se utilizar para decidir si dos detecciones se deben de fusionar.

También disponemos de otros tres ficheros, como son `data_obj_detect.py`, `data_track.py` y `utils.py`, que tienen diferentes funciones, pero no se profundizará en ellas debido a que el punto fuerte se encuentra en los dos ficheros anteriores para la optimización del modelo.

A. Detecciones de alta calidad

Se demuestra que se puede mejorar el rendimiento del seguidor de objetos de una manera muy sencilla considerando únicamente detecciones de alta calidad del detector. Esto es debido a que las detecciones de baja calidad, con puntuaciones muy bajas, tienden a ser falsos positivos o resultado del ruido por objetos pequeños o mal iluminados que producen confusión. Al filtrarlas, se reduce su cantidad, lo que mejora la precisión. También mejora la precisión al dar prioridad a detecciones donde el modelo tiene una mayor confianza.

Para ello, se ha estudiado la documentación [3] asociada a la clase *FasterRCNN*, la cual explicita 4 parámetros interesantes a los que se le puede hacer selección (*fine tuning*).

- `box_score_thresh` (float): Umbral mínimo de puntuación de clasificación para definir las detecciones que se van a considerar.
- `rpn_nms_thresh` (float): Umbral utilizado para filtrar respuestas redundantes con solapamiento significativo entre los cuadros delimitadores y la *RPN* (Red de Propuestas de Regiones).
- `box_nms_thresh` (float): Umbral utilizado para filtrar respuestas redundantes con solapamiento significativo entre los cuadros delimitadores.
- `rpn_score_thresh`: Umbral de puntuación de la *RPN*, utilizado para filtrar las propuestas menos probables antes del procesamiento posterior.

Estos 4 parámetros pueden ser ajustados para mejorar la detección de los objetos.

B. Algoritmo eficiente de asociación de datos

La estrategia de asociación empleada por la clase `TrackerIoUAssignment` puede ser vista como una forma de búsqueda codiciosa. En el seguimiento de objetos, la búsqueda codiciosa implica tomar decisiones óptimas localmente en cada paso sin considerar el óptimo global.

Para abordar este problema, se propone utilizar el algoritmo húngaro, una técnica utilizada para resolver el problema de asignación en el seguimiento de objetos. El algoritmo húngaro [2] busca iterativamente la asignación óptima empleando una combinación de operaciones matriciales y rutas de aumento. Esto garantiza encontrar la asignación globalmente óptima con el mínimo coste. En nuestro contexto, en el seguimiento de objetos, nos interesa utilizar esta técnica para maximizar la coincidencia entre las pistas existentes y las nuevas detecciones y así encontrar la asignación que maximiza la ganancia total.

Una explicación breve del funcionamiento de este algoritmo aplicado a nuestro contexto sería la siguiente:

- **Matriz de distancias:** En el seguimiento de objetos, las detecciones de objetos en un fotograma se pueden representar como las filas de una matriz, y las pistas de seguimiento (o las detecciones de objetos en el fotograma anterior) se pueden representar como las columnas de la matriz. Cada celda de la matriz contiene una medida de similitud (por ejemplo, *IoU* o distancia euclíadiana) entre una detección y una pista. Esta matriz se utiliza como entrada para el algoritmo húngaro. En el contexto del algoritmo húngaro, la matriz de distancias representa el costo de asignar cada detección a cada pista. El objetivo es minimizar este coste, lo que equivale a maximizar la similitud total entre las detecciones y las pistas. El algoritmo húngaro encuentra la asignación óptima mediante la búsqueda de rutas de aumento en la matriz de distancias, lo que garantiza que se encuentre la solución globalmente óptima para el problema de asignación de datos.
- **Ampliación de la matriz de costes.** El algoritmo húngaro asigna un único track a cada detección y a todas las detecciones. Sin embargo, podría suceder que el coste dejando algunas detecciones anteriores sin asignar fuera menor, por lo que el algoritmo húngaro no nos sirve para encontrar la solución óptima considerando también esto. Para solucionar este problema, se recurre a extender la matriz de costes tomando en consideración también las detecciones faltantes y las detecciones falsas, de tal modo que pueda haber así detecciones sin asignar y detecciones previas que han dejado de aparecer en el siguiente frame. El único problema con esta aproximación es que no conocemos los costes verdaderos de obtener una falsa detección o una detección faltante, por lo que se fijan mediante la experiencia.

- **Aplicación del algoritmo húngaro:** Una vez que se ha realizado la cobertura inicial, el algoritmo encuentra la mejor asignación existente considerando que puede haber falsas detecciones y detecciones faltantes.

III. IMPLEMENTACIÓN

A. Detecciones de alta confianza

Para la inclusión de únicamente las detecciones de alta confianza se han modificado en el propio notebook ciertas clases dadas donde se ejecuta toda la práctica. Se considera que esto aumenta la legibilidad del código a pesar de disminuir su robustez, pero lo mantenemos así porque resultaría difícil encapsular todas estas funciones en los archivos .py dados.

Así, se han modificado los siguientes aspectos de la clase *FRCNN-FPN* [3]:

- *box_score_thresh*: un valor igual a 0.81040990.
- *box_nms_thresh*: un valor igual a 0.5.
- *rpn_nms_thresh*: Un valor igual a 0.7.
- *rpn_score_thresh*: Un valor igual a 0.0

, donde el valor de *box_score_thresh* se han decidido de manera experimental, haciendo una selección (*tuning*) del parámetro con el conjunto de prueba (*test*) de manera que obtenga el mayor valor posible de la métrica *F1-score*, que combina la precisión (*precision*) y la exhaustividad (*recall*) en una única medida. Este parámetro tiene una gran importancia sobre el rendimiento del modelo debido a su función al filtrar detecciones con una confianza baja.

B. Forma eficiente de asociación de datos

Para la implementación del algoritmo húngaro se ha creado la clase *TrackerIoUAssignment_hungarian* heredando de la clase *Tracker* igual que en la clase *TrackerIoUAssignment* y se modifica la función *data_association*:

- **Cálculo de la distancia basada en *IoU*:** Se calcula la distancia entre las pistas existentes y las nuevas detecciones utilizando la Intersección sobre la Unión *IoU*. Un *IoU* más alto indica una mejor coincidencia entre una detección y una pista, por lo que una distancia más alta reflejaría una mejor similitud. Para esta parte se utiliza la función *distances.iou_matrix*, de la librería *motmetrics*, en el cual asignamos al parámetro *max_iou* el valor de 0.5, lo cual significa que se considera una superposición significativa entre dos cajas delimitadoras si existe más de un 50 % de superposición. Por último, se extiende esta matriz para poder aplicar el algoritmo húngaro con unos costes determinados.
- **Solución del problema de asociación:** Después de calcular los costes entre todas las posibles combinaciones de detecciones y pistas, se utiliza la función *linear_sum_assignment* de la librería *Scipy* para resolver el problema de asignación de datos. Esta función utiliza el algoritmo húngaro para encontrar la asignación óptima que al minimiza los costes totales.

- **Obtención de Asignaciones:** Despues de resolver el problema de asignación, *linear_sum_assignment* devuelve dos matrices de índices: *row_indices* y *col_indices*, las cuales contienen los índices de las pistas de seguimiento y las detecciones de objetos, respectivamente, que forman las asignaciones óptimas encontradas por el algoritmo húngaro.

C. Manejo de detecciones faltantes y de detecciones falsas

Este cambio se ha añadido en la clase *TrackerIoUAssignment_hungarian*, donde se ha añadido la función *extend_cost_matrix* para ello y haciendo los cambios pertinentes en la función *data_association*. La implementación para la extensión de la matriz de costes para añadir detecciones falsas y detecciones faltantes, lo cual es algo necesario para la ejecución del algoritmo húngaro, ha sido la siguiente:

- **Creación de la matriz extendida de costes:** Para ello se ha generado la función *extend_cost_matrix* que toma como entrada la matriz de costes (distancias basadas en *IoU*) y dos umbrales, *mis_det_thres* que se utilizará para llenar la matriz diagonal principal de esta parte de la matriz de costes. Al asignar un valor diferente de 1 (en este caso *mis_det_thres*) en la diagonal principal, se penaliza específicamente las asociaciones donde las detecciones no se asocian correctamente con una pista en el fotograma anterior, lo que modela las detecciones perdidas. Y *false_det_thres* que se utiliza igual que el anterior umbral. En este contexto, una falsa detección ocurre cuando una detección nueva se asocia incorrectamente con una pista existente en el fotograma anterior por lo que al establecer un valor diferente en la diagonal principal se penalizan específicamente estas asociaciones incorrectas. Por último, se concatenan estas matrices de costes para dar la matriz que puede emplear el algoritmo húngaro.
- **Bucle a través de los índices de asignación:** Se itera a través de los índices de asignación obtenidos de la matriz de asignación. Estos índices representan las asignaciones entre detecciones en el fotograma actual y tracks en el fotograma anterior.
- **Actualización de tracks existentes:** Si *i* y *j* son menores que *row_number* y *col_number*, respectivamente, esto significa que hay una asignación válida entre una detección anterior y una detección actual. En este caso, se actualiza la posición y la puntuación del track correspondiente con la nueva detección y puntuación. *row_number* representa el número de las detecciones en el fotograma actual y *col_number* representa el número de pistas detectadas en fotograma anteriores que se están considerando para asociarse con las detecciones actuales.
- **Adición de tracks nuevos:** Si *i* es menor que *row_number*, pero *j* es mayor o igual que *col_number*, esto indica una falsa detección. Manejamos esto suponiendo que siempre se trata de un objeto

nuevo que ha aparecido en la imagen. Así, se añade correspondientemente la nueva detección y puntuación a las listas `new_boxes` y `new_scores`.

- **Eliminar tracks ausentes:** Si i es mayor o igual que `row_number`, pero j es menor que `col_number`, esto indica una detección que falta. Siempre vamos a suponer que se trata de que el objeto ha abandonado la imagen. Se añade el `id` del track correspondiente a la lista `track_ids_to_remove`
- **Eliminar y añadir tracks:** Una vez completado el bucle, se eliminan los tracks marcados en `track_ids_to_remove` y se añaden nuevos tracks con las nuevas detecciones y puntuaciones almacenadas en `new_boxes` y `new_scores`.

D. Generación de vídeo

Por último, la función `export_to_mp4` toma todas las imágenes de cada vídeo etiquetadas y las junta en un vídeo en formato `.mp4`, que muestra los vídeos con las detecciones de las pistas de seguimiento superpuestas.

- 1) `plot_confusion_matrix(y_pred_proba, y_truth, threshold=0.5)`. Como parámetros de entrada recibe `y_pred_proba`, probabilidades predichas por el modelo y se convierte en etiquetas de clase binaria utilizando el umbral especificado; `y_truth`, referencia para calcular la matriz de confusión comparando estas etiquetas reales con las etiquetas predichas y `threshold`. Esto permite ajustar la sensibilidad del modelo al cambiar el punto de corte para clasificar las muestras. La función convierte las probabilidades predichas en etiquetas binarias según el umbral, calcula la matriz de confusión basada en las etiquetas verdaderas y las etiquetas predichas, y luego visualiza esta matriz utilizando un mapa de colores.
- 2) `plot_precision_recall(y_pred_proba, y_truth)`. Como parámetros de entrada tiene `y_pred_proba`, contiene las probabilidades predichas por el modelo y `y_truth`, contiene las etiquetas reales (verdaderas) de las muestras. Esta función visualiza el rendimiento del modelo en términos de precisión y recall, encuentra el umbral óptimo basado en la F1-score, y muestra la matriz de confusión para ese umbral óptimo.
- 3) `calcular_f1_max(y_truth, y_proba)`. Utiliza los mismos parámetros de entrada que los anteriores métodos, y la función encuentra y devuelve el puntaje F1 más alto posible y el umbral correspondiente a ese puntaje, usando las curvas de precisión (*precision*) y exhaustividad (*recall*).

E. Ejecutando el código

Para compilar el código simplemente se deberán ejecutar en orden las celdas del notebook adjunto en la entrega. Se ha ejecutado con recursos locales, motivo por el que las celdas correspondientes a la descarga del modelo base están comentadas.

IV. EXPERIMENTACIÓN

A. Detector de objetos

Este experimento tiene como objetivo evaluar un modelo de detección de objetos previamente entrenado, específicamente una variante del modelo *Faster R-CNN* con una red de características piramidales (*FRCNN_FPN*), utilizando un conjunto de datos de prueba. El proceso de evaluación comienza con la carga del modelo y su configuración para la evaluación. Una vez que el modelo está listo, se procede a seleccionar el conjunto de datos de prueba, en este caso, las secuencias del conjunto **MOT16**.

El paso crucial del experimento implica la evaluación del modelo en el conjunto de datos de prueba. Esto implica la generación de predicciones del modelo sobre las imágenes de prueba y la comparación de estas predicciones con las anotaciones *ground truth*, que representan las ubicaciones reales de los objetos en las imágenes. La comparación se realiza para calcular métricas de evaluación importantes, como la *precisión* y el *recall* del modelo.

Una vez que se han calculado estas métricas, el experimento busca encontrar y optimizar el umbral correcto para la puntuación de clasificación del modelo. Esto se logra mediante la visualización de la curva *precision-recall*, que muestra cómo varían la *precisión* y el *recall* para diferentes umbrales de puntuación. El objetivo final es seleccionar el umbral que maximice simultáneamente la *precisión* y el *recall* del modelo, lo que indica un equilibrio óptimo entre la capacidad de clasificación y la capacidad de detección del modelo.

Al finalizar el experimento, se espera haber identificado el umbral óptimo que permite al modelo realizar predicciones precisas y exhaustivas sobre la presencia y ubicación de objetos en las imágenes de prueba.

Tras este experimento se obtiene que para el umbral `box_score_thresh` se obtiene un valor de 0.81 y con ello se logra una mejora en la puntuación *F1-score* hasta un valor de 95.57

B. Algoritmo húngaro

Para el experimento para comprobar el funcionamiento del tracker con el algoritmo húngaro y el añadido de la matriz de costes extendida, simplemente se ha ejecutado el tracker y su evaluación con el cambio en el algoritmo de asignación y se ha observado los resultados, lo cual ha mostrado un gran mejora en la asignación de detección y en las métricas de test.

V. DATOS UTILIZADOS

Se ha usado el conjunto de datos **MOT16** [1] que es empleado como una referencia para la evaluación de modelos seguimiento de objetos múltiples.

Es un conjunto de vídeos etiquetados con seguimiento de múltiples objetos en situaciones realistas, tanto con cámara fija como móvil. Está dividida en 7 secuencias para entrenamiento y 7 secuencias de test.

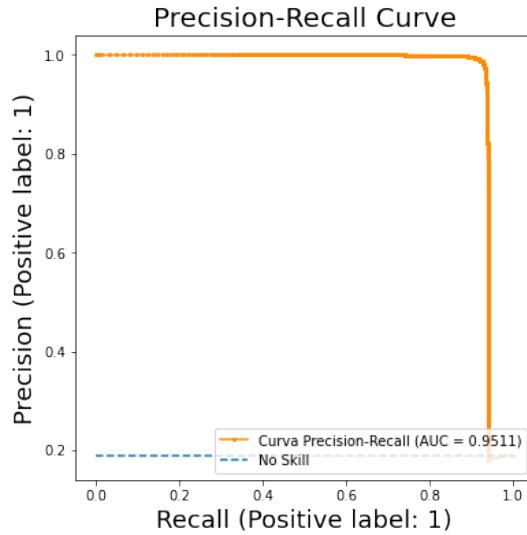


Fig. 1. Curva precisión-exhaustividad para diferentes umbrales de `box_score_thresh`.

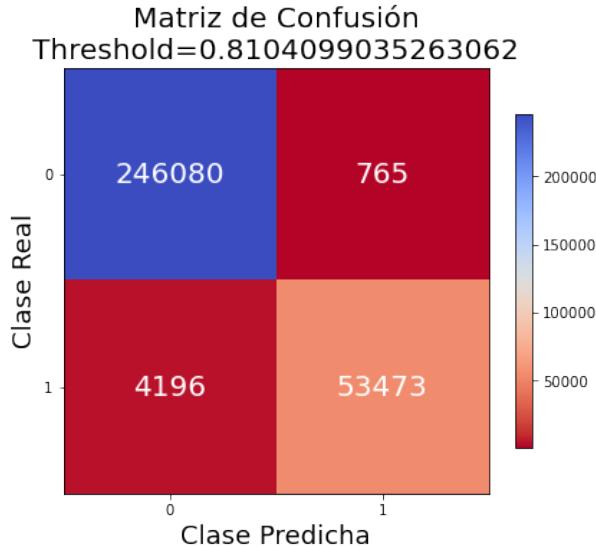


Fig. 2. Matriz de confusión para el umbral óptimo de `box_score_thresh` igual a 0.81040990.

Para entrenamiento se ha utilizado el siguiente subconjunto de datos:

- **MOT16-02:** Secuencia capturada alrededor de una plaza con gente caminando
- **MOT16-04:** Secuencia capturada en una calle peatonal de noche, con un punto de vista elevado.
- **MOT16-05:** Secuencia capturada en una calle a través de una plataforma móvil.
- **MOT16-09:** Secuencia capturada en una calle peatonal desde un ángulo cerrado.

y en test se han utilizado los subconjunto de datos:

- **MOT16-10:** Secuencia capturada de una calle peatonal por la noche con una cámara en movimiento.
- **MOT16-11:** Secuencia capturada de una cámara en

movimiento hacia adelante en un concurrido centro comercial.

- **MOT16-13:** Secuencia capturada desde un autobús a una intersección concurrida.

VI. RESULTADOS Y ANÁLISIS

En cuanto a la comparación de la evaluación del detector de objetos base y con los umbrales modificados, se observa una pequeña mejora respecto a las métricas usadas en el método de evaluación.

```
AP: 84.7773 %. Prec: 94.5962 %. Rec: 89.6079 %. TP: 51676. FP: 2952. F1-score: 92.0345 %.
CPU times: user 4min 57s, sys: 15.9 s, total: 5min 13s
Wall time: 5min 14s
```

Fig. 3. Detector de objetos base.

```
AP: 91.4213 %. Prec: 98.5895 %. Rec: 92.7223 %. TP: 53472. FP: 765. F1-score: 95.5659 %.
CPU times: user 4min 59s, sys: 18.1 s, total: 5min 17s
Wall time: 5min 20s
```

Fig. 4. Detector de objetos con ajuste fino de `box_score_thresh`.

En cuanto a la mejora en el `tracker`. Primero, se ejecuta el algoritmo de búsqueda base con *IoU*. Luego, se vuelve a ejecutar el código usando el algoritmo con las mejoras implementadas y explicadas anteriormente en el mismo conjunto de datos para comparar los resultados. También se añaden los resultados obtenidos tras las mejoras producidas por el `tracker` tras añadir los umbrales ajustados. Se comparan las métricas devueltas por el método `evaluate_mot_metrics` perteneciente al archivo `utils.py` dentro de la carpeta `trackers`.

```
Runtime for all sequences: 259.7 s.
IDF1 IDP IDR Rcll Prcn GT MT PT ML FP FN IDs FM MOTA MOTP
MOT16-13 46.5% 64.2% 36.5% 45.1% 79.2% 110 28 34 48 1375 6393 82 124 32.6% 0.134
MOT16-11 49.0% 54.1% 44.8% 55.8% 67.5% 75 15 32 28 2542 4166 20 39 28.7% 0.080
MOT16-10 35.3% 48.6% 31.3% 42.0% 54.5% 57 13 23 21 4504 7449 69 120 6.4% 0.138
OVERALL 42.9% 51.3% 36.8% 46.9% 65.4% 242 56 89 97 8421 18008 171 283 21.6% 0.118
```

Fig. 5. Tracker base

```
Runtime for all sequences: 255.2 s.
IDF1 IDP IDR Rcll Prcn GT MT PT ML FP FN IDs FM MOTA MOTP
MOT16-13 48.5% 72.4% 36.4% 43.8% 87.0% 110 31 29 50 764 6548 71 76 36.6% 0.127
MOT16-11 52.5% 58.0% 48.0% 55.1% 66.7% 75 15 29 31 2600 4236 19 24 27.4% 0.075
MOT16-10 36.3% 46.5% 29.8% 48.9% 63.8% 57 13 22 22 2976 7598 54 83 17.3% 0.134
OVERALL 45.1% 57.5% 37.1% 45.8% 71.0% 242 59 80 183 6348 18374 144 183 26.7% 0.112
```

Fig. 6. Seguidor de objetos con ajuste fino de `box_score_thresh` para el detector de objetos.

```
Runtime for all sequences: 258.3 s.
IDF1 IDP IDR Rcll Prcn GT MT PT ML FP FN IDs FM MOTA MOTP
MOT16-13 66.6% 72.1% 61.1% 83.8% 97.7% 110 78 27 5 227 1887 321 182 79.1% 0.130
MOT16-11 66.7% 75.8% 59.7% 78.2% 99.3% 75 39 28 8 50 2055 55 48 77.1% 0.076
MOT16-10 52.2% 58.4% 47.1% 78.7% 97.6% 57 35 21 1 253 2737 266 222 74.6% 0.144
OVERALL 61.2% 68.6% 55.7% 80.3% 98.1% 242 152 76 14 530 6679 642 452 76.9% 0.121
```

Fig. 7. Seguidor de objetos con detector mejorado y con algoritmo húngaro

Se puede apreciar que hay una gran mejora respecto al modelo base en torno a todas las métricas y sin un coste computacional mayor al añadir un algoritmo óptimo para la asignación de detecciones y un umbral para las detecciones obtenidas en el detector de objetos se obtiene una gran

mejora. Y una pequeña mejora al ajustar los umbrales del *Faster-RCNN*.

De forma visual, esto se puede observar con la comparación de dos frames tras haber ejecutado cada uno de los trackers por separado.



Fig. 8. Frame de **MOT16-10**. Modelo base.



Fig. 9. Frame de **MOT16-10**. Modelo final mejorado.

Ambos algoritmos funcionan perfectamente cuando el objeto es muy claro, pero cuando el objeto es más pequeño y difícil de distinguir, el modelo mejorado supera en cuanto a precisión al modelo base. Se observa que una de las personas (ID-52 del **MOT16-10** modelo mejorado) del fondo se pierde usando el tracker del modelo base, por lo que el modelo mejorado logra mejores resultados en el seguimiento. También en los frames del **MOT16-13** en el modelo base se observa que la caja delimitadora con ID22 no está asignada a una persona. Se recomienda para observar bien la mejora, mirar los vídeos adjuntados en la entrega en el directorio *output* para observar la gran mejora que tiene.

Un punto débil que se ha observado del modelo mejorado es que, si se accede a cualquiera de los vídeos que se adjuntan con la entrega, en momentos en los que se producen cierta cantidad de oclusión, el tracker no es capaz de hacer el seguimiento correctamente ya que desaparece. Y aunque se obtengan mejores resultados y en todas las métricas se obtiene un resultado totalmente aceptable, se puede llegar a mejorar con creces sobre todo las métricas *IDF1*, *IDP* e *IDR*. Un aspecto curioso y punto débil de los dos modelos es que en las imágenes del **MOT16-11**, los dos modelos son



Fig. 10. Frame de **MOT16-11**. Modelo base

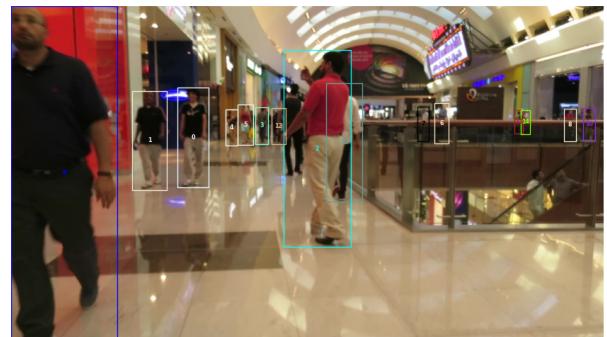


Fig. 11. Frame de **MOT16-11**. Modelo final mejorado.



Fig. 12. Frame de **MOT16-13**. Modelo base



Fig. 13. Frame de **MOT16-13**. Modelo final mejorado.

incapaces de detectar a las dos personas que están subiendo las escaleras, las cuales se observan de forma clara en la imagen, lo cual creemos que es debido a un problema de oclusión con el cristal y su reflejo.

VII. CONCLUSIONES

A la vista de los resultados, se concluye la superioridad del algoritmo húngaro a la hora de asignar las detecciones actuales a las anteriores. Como futura mejora se plantea el ajuste fino de todos los parámetros restantes de la clase *FasterRCNN* de la librería *torchvision*.

Como futuro trabajo, para solucionar el problema comentado en los resultados sobre la pérdida de seguimiento como consecuencia de la oclusión de los objetos de la imagen, se podría plantear la predicción de las trayectorias de los objetos, que incluso se podrían unir con un *filtro de Kalman* [5] para reducir la incertidumbre en la estimación.

VIII. LOG DE TIEMPOS

Los tiempos aproximados empleados para la realización de las distintas partes del proyecto han sido los siguientes:

- Análisis y ejecución del modelo base: 4 horas.
- Diseño de la mejora: 10 horas.
- Implementación y ejecución de la mejora: 13 horas.
- Redacción de la memoria: 8 horas.

REFERENCES

- [1] <https://motchallenge.net/data/MOT16/>
- [2] <https://thinkautonomous.medium.com/computer-vision-for-tracking-8220759eee85>
- [3] https://github.com/pytorch/vision/blob/main/torchvision/models/detection/faster_rcnn.py
- [4] R.E. Burkard, M. Dell'Amico, S. Martello: *Assignment Problems* (Revised reprint). SIAM, Philadelphia (PA.) 2012.
- [5] Bewley, A., et al. *Simple online and realtime tracking*. IEEE ICIP 2016.