UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

ESCOLA TÉCNICA SUPERIOR DE ENXEÑARÍA

Detección de objetos mediante *few-shot*

Autor:
Pablo García Fernández

Directores:
Manuel Mucientes Molina
Daniel Cores Costa

Grao en Enxeñaría Informática

Julio 2021

Trabajo de Fin de Grado presentado en la Escuela Técnica Superior de Ingeniería de la Universidad de Santiago de Compostela para la obtención del Grado en Ingeniería Informática

D. Manuel Mucientes Molina, Profesor del Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela, y **D. Daniel Cores Costa**, Investigador predoctoral en el Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS),

INFORMAN:

Que la presente memoria, titulada *Detección de objetos mediante few-shot*, presentada por **D. Pablo García Fernández** para superar los créditos correspondientes al Trabajo de Fin de Grao de la titulación del Grado en Ingeniería Informática, fue realizada bajo nuestra dirección en el Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela.

Y para que así conste a los efectos oportunos, expiden el presente informe en Santiago de Compostela, a 25 de junio de 2020:

El director,	El codirector,	El alumno,
MUCIENTES MOLINA MANUEL - 44810714K Firmado digitalmente por MUCIENTES MOLINA MANUEL - 44810714K Fecha: 2021.06.25 16:59:35 +02'00'	CORES COSTA DANIEL - 35490562K Firmado digitalmente por CORES COSTA DANIEL - 35490562K Fecha: 2021.06.25 14:32:12 +02'00'	
Manuel Mucientes Molina	Daniel Cores Costa	Pablo García Fernández

Agradecimientos

- Gracias a mi familia, y en especial a mi hermano Miguel, por estar siempre a mi lado.
- A *futfi* y a Alex-kohai por hacer esta etapa de mi vida mucho más divertida.
- A mis amigos de siempre Adrián, Víctor, Bello, Antón, Xoel y Marcos.
- A mis directores Manuel Mucientes y Daniel Cores por ayudarme en todo momento.

Resumen

Los últimos avances en detección de objetos en imágenes están impulsados, principalmente, por el uso de redes neuronales convolucionales sobre grandes conjuntos de datos etiquetados. Sin embargo, cuando estos datos de entrenamiento son limitados, el rendimiento de los detectores se deteriora considerablemente. Para solventar este problema surge el aprendizaje *few-shot*; redes con capacidad para detectar objetos de categorías nunca vistas a partir de un reducido número de ejemplos etiquetados. En este trabajo, partiendo de la red de detección *few-shot* FewX [1], se ha analizado experimentalmente la capacidad de generalización y desempeño de estas arquitecturas. Así se ha observado que, pese a la habilidad para detectar objetos nunca vistos, estas aproximaciones todavía tienen un amplio margen de mejora.

Además, una vez adquirida una profunda comprensión acerca del problema, se ha modificado FewX. El principal cambio es la introducción de una *Feature Pyramid Network*. Los experimentos demuestran que nuestra versión de la red supera a FewX en todas las métricas.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introdución

1.1. Problemática y motivación

La **visión por computador** es un campo interdisciplinar que tiene por objetivo desarrollar sistemas capaces de adquirir, analizar e interpretar imágenes procedentes del mundo real. Haciendo una analogía con la biología, lo que se pretende es crear sistemas autónomos con capacidad para desempeñar algunas de las tareas que el sistema visual humano puede realizar [5].

Desde sus inicios en la década de los sesenta [6], muchos son los avances que se han realizado. Probablemente uno de los más relevantes sea el uso de algoritmos de aprendizaje automático, particularmente **redes neuronales convolucionales** (CNNs), para extraer las características de una imagen de entrada. Estas características pueden ser utilizadas, posteriormente, para reconocer objetos, clasificarlos, detectarlos, etc.

A día de hoy, todavía quedan una gran variedad de problemas en este ámbito sin resolverse satisfactoriamente. Uno de los más importantes, es la inmensa cantidad de datos etiquetados necesarios para entrenar redes convolucionales. Estos datos de entrenamiento son **limitados**, **costosos de obtener** (etiquetado manual) y **condicionan fuertemente el desempeño** final del modelo construido.

A diferencia de los humanos, que tenemos capacidad para detectar características visuales a partir de unos pocos ejemplos dados, las CNN sufren una caída de rendimiento significativa cuando los datos son escasos. Así, un niño es capaz de comprender y generalizar un nuevo concepto (p. ej. "coche") visto un ejemplo; mientras que las mejores arquitecturas de CNN necesitan cientos o miles de ellos.

Centrando la atención en el problema de detección de objetos, es muy difícil

que un algoritmo tradicional (como *Faster R-CNN*, *YOLO* u otros) sea capaz de reconocer y localizar objetos de categorías nunca vistas sin un exhaustivo reentrenamiento con abundantes ejemplos de la nueva clase a detectar. Este problema se acentúa en entornos de escasez de datos donde puede resultar muy complicado o incluso imposible obtener imágenes correctamente etiquetadas para entrenar. En estos casos es imprescindible el uso eficiente de la poca información disponible para generar conocimiento relevante.

Surgen de este modo técnicas de aprendizaje **basadas en *few-shot***. En esencia se trata de algoritmos de aprendizaje automático que infieren resultados a partir de un **reducido número de ejemplos de entrenamiento**. En nuestro caso particular exploraremos el problema de la detección de objetos *few-shot*. En el mismo se busca obtener un modelo con capacidad para detectar objetos de categorías nunca vistas a partir de un conjunto reducido de ejemplos etiquetados.

En este aspecto una aproximación que ha dado buenos resultados en los últimos años, y en la que nos centraremos en este Trabajo Fin de Grado, es el **meta-aprendizaje** a través de ***matching networks***. Se trata de redes de pesos compartidos en las que cada una de las distintas ramas es alimentada respectivamente, con k **imágenes de soporte** (patrón del objeto a detectar) y una **imagen de consulta**. El objetivo es que el sistema *few-shot* aprenda automáticamente **métricas de similitud** entre ambos elementos independientemente de la categoría. Es decir, no se persigue la capacidad de la red para detectar objetos pertenecientes a una categoría particular, sino la habilidad para **relacionar objetos de soporte con consultas**. Para ello, primero se entrena al modelo bajo un conjunto de clases base con suficientes ejemplos para, posteriormente, evaluar su desempeño sobre otro conjunto de clases nuevas con pocas muestras.

En este trabajo, partiendo de la red de detección FewX propuesta en [1], analizaremos cómo influye en el desempeño del modelo la variación en el número de imágenes de soporte y sus categorías. Esto nos permitirá evaluar la capacidad de generalización de este tipo de redes. Finalmente, una vez adquirida una profunda comprensión acerca de este tipo de arquitecturas, propondremos e implementaremos una serie de mejoras.

1.2. Hipótesis a probar

Las hipótesis a probar son:

1. Testear sobre clases muy parecidas a las de entrenamiento sesga positivamente los resultados obteniendo valores elevados. Por el contrario, sobre clases muy distintas los resultados son mucho peores.

2. Un mayor número de imágenes de soporte en una categoría conlleva una mejor comprensión de las características del objeto a detectar y, consecuentemente, una mayor precisión del modelo.

3. Reentrenar la red con la información de soporte de las clases nuevas (*fine-tuning*) optimiza los pesos del modelo adaptándolos a las categorías de test y mejorando, de esta forma, el desempeño de la red.

4. La introducción de una *Feature Pyramid Network* (FPN) permite detectar objetos en diferentes resoluciones del mapa de características, haciendo más robusta la detección de objetos de diferentes tamaños.

1.3. Objetivos

El objetivo general de este Trabajo Fin de Grado es **estudiar el problema del aprendizaje *few-shot*** en el marco de la detección de objetos, concretamente sobre la red FewX, e **implementar y validar las mejoras** que se hagan sobre la misma. Se espera que, una vez concluido, hayamos analizado detalladamente la capacidad de generalización de este tipo de redes, y mejorado el desempeño de FewX a través de los cambios implementados.

De este modo, para la consecución de este objetivo general se proponen los siguientes **5 subobjetivos:**

1. Analizar la red FewX [1] y comprender pormenorizadamente su funcionamiento interno.

2. Diseñar y realizar un estudio experimental que nos permita evaluar las capacidades de la red FewX. Implementar las modificaciones y mecanismos necesarios para poder llevarlo a cabo correctamente.

3. Analizar los resultados obtenidos examinando el comportamiento de la red para soportes de distintos tamaños, clases y técnicas de reentrenamiento. Identificar los puntos débiles de la arquitectura que sean susceptibles de posibles mejoras.

4. Introducir una FPN e implementar el conjunto de mejoras previamente identificadas.

5. Evaluar el desempeño de la nueva versión de la red respecto a la red original y a otras aproximaciones del estado del arte.

1.4. Estructura de la memoria

Esta memoria sigue la **estructura tipo A**, propuesta en el reglamento del Trabajo de Fin de Grado de Ingeniería Informática y aprobada por el *Consello de Goberno* el 19 de junio de 2020:

En el **capítulo 2** se revisa el estado actual del problema, las técnicas existentes y las diferencias y similitudes de nuestro trabajo con ellas.

En el **capítulo 3** se define formalmente el problema, se describe el funcionamiento básico de FewX y se especifican los cambios y mejoras introducidas sobre la red original. También se analiza el material necesario para llevar a cabo el trabajo presentado y se detallan los diferentes métodos utilizados para realizar el estudio experimental.

En el **capítulo 4** se presentan las pruebas realizadas y los resultados obtenidos.

En el **capítulo 5** se discuten los resultados obtenidos y se relacionan con las hipótesis de partida. También se comparan los resultados obtenidos por nuestra versión de la red con los resultados de otras aproximaciones del estado del arte.

En el **capítulo 6** se recogen las conclusiones del trabajo y las posibles ampliaciones.

Capítulo 2

Estado del conocimiento

2.1. Detección de objetos

Dentro de la visión por computador, la detección de objetos es uno de los problemas más ampliamente estudiados. Consiste en localizar los múltiples objetos que pueden aparecer en una imagen y establecer la categoría de cada uno de ellos (figura 2.1).

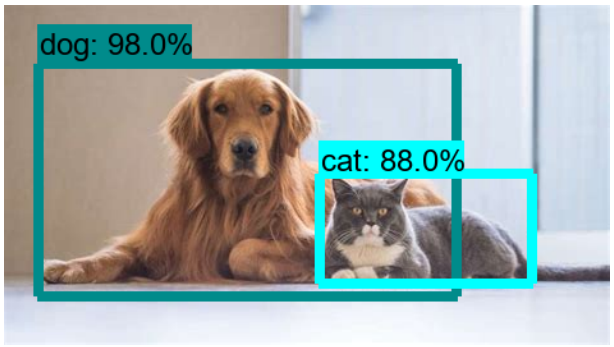


Figura 2.1: Detección de objetos. Imagen extraída de [57]

En sus inicios, la extracción de características de una imagen se realizaba a través de métodos predefinidos como SIFT (*Scale-invariant feature transform*) o HOG (*Histogram of Oriented Gradients*). Posteriormente, las características extraídas se utilizaban como entrada en algún algoritmo de detección. Con el surgimiento de las CNNs el paradigma cambió y este tipo de aproximaciones se convirtieron en las dominantes.

Los detectores de objetos basados en CNN pueden dividirse en dos categorías generales: **libres de propuestas** y **basados en propuestas**. La primera línea de trabajo sigue una estrategia de entrenamiento de **una única etapa** y no genera explícitamente regiones de interés. Por este motivo son significativamente

más rápidos que los segundos. Su ejemplo más representativo es **YOLO** [2]. Este utiliza una única ejecución de red convolucional para realizar directamente las predicciones de la clase y del cuadro delimitador. Otro ejemplo es SSD, [3] que mejora a YOLO utilizando cajas predeterminadas (*anchors*) para ajustarse a diversas formas de objetos.

Por contra, los detectores basados en propuestas están conformados por **dos etapas**. En la primera, se generan regiones de interés (porciones de la imagen con altas probabilidades de contener objetos a detectar). A continuación, cada región es refinada y clasificada. En sus inicios, arquitecturas como **R-CNN** [23] generaban estas regiones a través de algoritmos de **búsqueda selectiva** [26] aplicados sobre la imagen de entrada original. A continuación, cada región era procesada por una red convolucional (CNN), obteniendo como salida la clase del objeto predicho y su cuadro delimitador. Los dos principales problemas de esta aproximación eran: la enorme cantidad de **tiempo** que costaba procesar una imagen (tantas ejecuciones de CNN como regiones propuestas) y la dificultad para generar propuestas válidas utilizando un **algoritmo preestablecido** (que no se puede aprender).

Con la llegada de **Fast R-CNN** [24] se soluciona el primero de los problemas. En esta nueva arquitectura las regiones de interés precalculadas ya no son procesadas independientemente por una CNN, sino que se proyectan sobre el mapa de características previamente obtenido de procesar la imagen (una única ejecución de CNN). Cada una de las propuestas puede ser de distinto tamaño, por lo que se vuelve inevitable incluir una capa de alineamiento (***RoI Pooling***) cuya función sea uniformar las dimensiones de las regiones a un tamaño constante (de forma que puedan ser suministradas, posteriormente, a una capa completamente conectada).

A pesar de las mejoras introducidas por esta arquitectura, seguía siendo necesario un algoritmo de búsqueda selectiva para obtener regiones de interés. **Faster R-CNN** [25] surge para resolver este problema. En esta arquitectura, ya no se emplea un algoritmo de búsqueda selectiva para identificar regiones de interés, sino que una red integrada en la propia CNN, llamada ***Region Proposal Network*** (RPN), se encarga de generar estas propuestas. FewX sigue una aproximación en dos etapas.

2.2. *Few-shot learning*

Consiste en métodos de aprendizaje automático que infieren resultados a partir de un **número muy reducido de ejemplos etiquetados**. El objetivo es hacer que un modelo alcance un rendimiento significativo cuando se dispone,

únicamente, de un conjunto limitado de información etiquetada (D_{nuevo}) y de un conocimiento previo (información obtenida de otros problemas, dominios, tareas, etc.). Si D_{nuevo} contiene N categorías con K ejemplos en cada una de ellas, el problema es denominado *N-way-K-shot*.

En [27], en función del aspecto que se mejora utilizando el conocimiento previo, se propone la siguiente taxonomía de métodos de aprendizaje en dominios con pocos datos:

- **Datos**. Estos métodos emplean el conocimiento previo para aumentar D_{nuevo}, incrementando el número de patrones etiquetados. Un ejemplo de ello es el uso de técnicas de ***data augmentation*** [7], [8], [9]. Pese a ser una aproximación simple tiene dos problemas fundamentales. Por un lado, las mejoras no son significativamente relevantes. Por otro, la política de aumento de datos suele estar hecha a medida para conjuntos de datos concretos, haciendo muy difícil su extrapolación a otros dominios.

- **Modelo**: Estos métodos utilizan el conocimiento previo para restringir la complejidad del problema a otro problema análogo, cuya resolución pueda generalizarse al original. Una solución es **transferir el conocimiento** desde una tarea de origen, donde los datos etiquetados son abundantes, a otra tarea destino, donde los datos son escasos [29]. Otra solución, cada vez más popular, es el **meta-aprendizaje sobre tareas invariantes**. Es decir, partiendo de un problema dividido en dos tareas relacionadas, el objetivo es meta-aprender de la primera una función general que pueda ser aplicable directamente a la segunda. Esta función puede consistir, entre otros, en: 1) una métrica de similitud (*matching network*) que permita relacionar una imagen de consulta con un conjunto reducido de imágenes etiquetadas [1]. 2) un optimizador de rápida adaptación. En [13] se propone para esto un LSTM (*Long Short-Term Memory*). 3) un predictor de parámetros. En [14] Learnet aprende automáticamente los parámetros del modelo en base a un solo ejemplo etiquetado.

- **Algoritmo**: Estos métodos emplean el conocimiento previo para alterar la estrategia de búsqueda de los parámetros θ que definen el modelo. Algunos ejemplos son: 1) usar un θ inicial aprendido de otras tareas y refinarlo empleando D_{nuevo} [10], [11]. 2) obtener, sobre otras tareas, un *meta-learner* que actúe como optimizador en el problema actual [12].

FewX puede incluirse dentro de la línea del **meta-aprendizaje** a través de ***matching networks***.

2.3. Detección de objetos *few-shot*

Muchos de los trabajos en este campo están basados en la estrategia de **transferencia de conocimiento**. Uno de los más representativos es LSTD [29]. En él se propone un modelo capaz de extrapolar el conocimiento de un gran conjunto de datos a otro conjunto más pequeño minimizando las diferencias entre ambos. Sin embargo, este método depende en gran medida del dominio del problema de origen y es difícil de extender a escenarios muy diferentes. TFA [34] y MPSR [33] mejoran esta línea de trabajo, pero siguen necesitando un ajuste fino de los parámetros para adaptar el modelo a los datos nuevos.

Una tendencia más reciente, y con mejores resultados, es el uso de métodos basados en el **meta-aprendizaje**. Estos poseen la capacidad de adaptarse fácilmente a datos nunca vistos a partir de un número reducido de muestras de soporte, sin necesidad de un ajuste forzoso. Siguiendo este paradigma se encuentran (entre otros) Meta-YOLO [28] y Meta-DETR [4], como detectores de una sola etapa; y Meta R-CNN [30], MetaDet[35] y FewX [1] como detectores de dos etapas (basados en Faster-RCNN).

Los detectores de dos etapas mencionados, dependen en gran medida de la calidad de las regiones propuestas por la RPN. Sin embargo, estas regiones son deficientes cuando los datos etiquetados son escasos. Para mitigar este problema, FewX meta-aprende una RPN basada en mecanismos de atención correlacionando los mapas de las imágenes de consulta y soporte. A pesar de ello el problema no desaparece. Por contra, Meta-DETR descarta el uso de mecanismos basados en regiones de interés y unifica el meta-aprendizaje de la localización y clasificación de objetos en un único módulo; así obtiene los mejores resultados del estado del arte.

En este trabajo emplearemos como punto de partida FewX, pues Meta-DETR es una aproximación muy reciente (marzo de 2021) y su código aún no está disponible.

Capítulo 3

Metodología

3.1. Definición del problema

Dados dos conjuntos de categorías C_{base} y C_{nuevas} de forma que $C_{base} \cap C_{nuevas} = \emptyset$; se persigue obtener un modelo *few-shot* con capacidad para detectar objetos pertenecientes a $C_{base} \cup C_{nuevas}$ [4], aprendiendo de un conjunto de datos base D_{base} con suficientes imágenes etiquetadas de C_{base}, y de otro conjunto de datos novedoso D_{nuevo} con *F-shot* imágenes por categoría en C_{nuevas}.

Si *F-shot* = 0 el modelo únicamente es entrenado sobre el conjunto de datos D_{base}. Si *F-shot* > 0 puede realizarse un segundo entrenamiento de *finetuning* sobre D_{nuevo}. Este reentrenamiento empleará un conjunto mucho más reducido de imágenes, es decir $|D_{base}| >> |D_{nuevo}|$.

Una vez entrenado, el modelo se evalúa recibiendo como entrada *K-shot* imágenes de soporte de la categoría $C_i \in C_{nuevas}$ y una imagen de consulta. El objetivo es detectar en la imagen de consulta todos los objetos que pertenecen a C_i.

En caso de haber realizado *finetuning* (reentrenar sobre D_{nuevo}) las primeras *F-shot* imágenes por categoría que integran el soporte de test son extraídas de D_{nuevo}. Habitualmente, *F-shot* es igual a *K-shot* (realizando *finetuning* sobre todo el conjunto de soporte de test).

Además, para que pueda ser considerado un problema few-shot, *K-shot* y *F-shot* deben ser valores muy pequeños, típicamente inferiores a 10 (aunque en la experimentación de este TFG probaremos valores significativamente superiores).

Para la construcción del modelo partimos de un problema dividido en dos tareas (T_b, T_f) relacionadas, donde T_f es la tarea *few-shot*. Esta solo se diferencia

de la primera en el número (reducido) de ejemplos etiquetados de los que se dispone para poder completarla satisfactoriamente. La tarea T_b tiene asociado el conjunto de datos D_{base}. La tarea T_f, el conjunto D_{nuevas}. Como ambas tareas consisten en aprender una métrica de similitud entre dos elementos (*matching networks*), se espera que el modelo construido sobre D_{base} funcione relativamente bien en la tarea few-shot (T_f). Dicho de otra forma, el objetivo es meta-aprender de T_b una función general δ que pueda ser aplicable directamente a T_f.

3.2. FewX

Nuestro punto de partida es la red de detección *few-shot* FewX descrita en [1]. Esta emplea aprendizaje supervisado para construir, de forma automática, funciones de similitud que permiten cuantificar la relación entre imágenes de soporte e imágenes de consulta.

La principal ventaja de FewX, y lo que la diferencia de otras aproximaciones similares, es la capacidad para detectar objetos de categorías nunca vistas a partir de un conjunto reducido de ejemplos etiquetados **sin necesidad de un reentrenamiento** del modelo. Es decir, una vez que la red fue entrenada sobre un conjunto suficientemente amplio de imágenes (consulta y soporte) etiquetadas (D_{base}), esta puede detectar objetos de categorías nunca vistas dados *k-shot* ejemplos de ellas (D_{nuevo}).

Para tal labor es necesario disponer de colecciones de datos etiquetados (*datasets*) que puedan ser utilizados para entrenar un modelo. Concretamente, estos datos de entrenamiento son de la forma $\{x_i, y_i\}_{i=1}^{N}$; donde N es el número de elementos, x son los datos de entrada e y las salidas esperadas [15]. En la figura 3.1 podemos observar visualmente el formato de entradas y salidas.

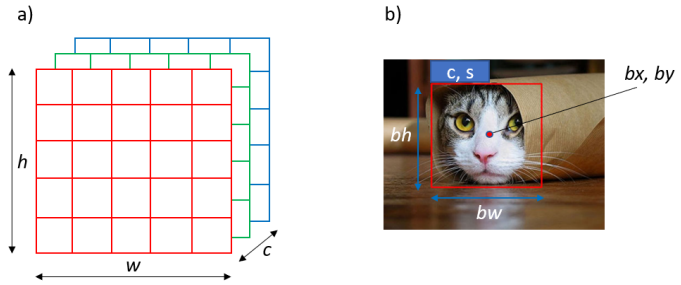


Figura 3.1: a) Tensor de entrada (x_i). b) Salida esperada (y_i). Imagen b) extraída parcialmente de [53]

En este caso **los datos de entrada** (x) son imágenes codificadas como tensores de rango 3 ($h \times w \times c$); siendo h la altura de la imagen, w la anchura y c el

número de canales.

Las salidas esperadas (y) son cinco elementos (c, bx, by, bh, bw); donde c señala la categoría del objeto predicho, (bx, by) determina el centro del cuadro delimitador, bh su altura y bw su anchura.

El entrenamiento de la red se realiza mediante tripletes de imágenes etiquetadas (q_c, s_c, s_n) en donde: q_c es una imagen de consulta escogida aleatoriamente, s_c , una imagen de soporte representativa de un objeto presente en q_c y s_n, una imagen de soporte representativa de un objeto no presente en q_c. Con esto, el objetivo que se persigue durante el entrenamiento es que el sistema aprenda tanto a emparejar objetos pertenecientes a la misma categoría, como a diferenciar objetos de categorías diferentes.

El núcleo de FewX es una red de detección en dos etapas ***Faster R-CNN*** [25]. Su arquitectura detallada se muestra en la figura 3.2. Puede observarse que está compuesta por tres unidades principales: una **red de pesos compartidos**, un **módulo de propuestas de regiones basado en mecanismos de atención** y un **detector de relaciones múltiple**.

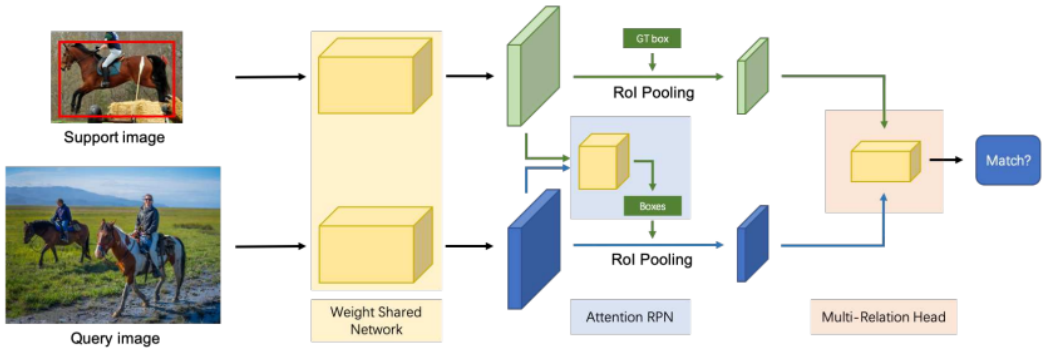


Figura 3.2: Arquitectura de la red. Imagen extraída de [1]

3.2.1. Red de pesos compartidos

La red de pesos compartidos está constituida por un conjunto de ramas. Concretamente, una para el conjunto de imágenes de soporte y otra para la imagen de consulta. El objetivo de este módulo es procesar ambos elementos y extraer sus respectivos mapas de características. Cuando una categoría en el soporte contiene varios ejemplos se toma como referencia el mapa promedio.

FewX emplea la red convolucional **ResNet-50** como extractor de características (para ambas ramas). Esta consiste en un conjunto de filtros aplicados a una imagen de entrada con el objetivo de obtener un mapa de características que recoja solo la información semántica más importante. Es decir, las características extraídas preservan únicamente los rasgos relevantes, eliminando aquellos que sean innecesarios para la comprensión de la imagen [16]. La arquitectura de ResNet-50 se recoge en la figura 3.3 y tabla 3.1.

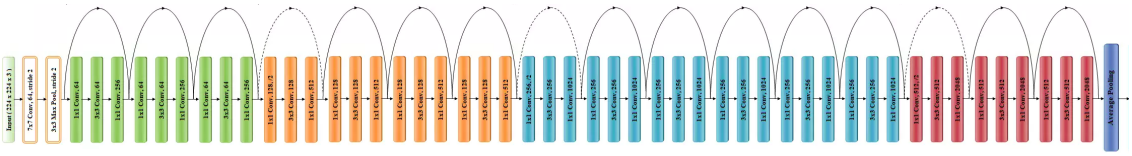


Figura 3.3: Arquitectura ResNet-50. Imagen extraída de [54]

layer name	**output size**	**50-layer**
conv1	112×112	7×7, 64, stride 2
conv2_x (verde)	56×56	3×3 max pool, stride 2
		$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x (naranja)	28×28	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$
conv4_x (azul)	14×14	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$
conv5_x (rojo)	7×7	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool

Tabla 3.1: Arquitectura detallada de ResNet-50.

A grandes rasgos puede observarse la existencia de dos tipos de capas, que se repiten a lo largo de toda la estructura y que conforman los bloques básicos de la red: **convolución** y ***pooling***.

En la capa de convolución un conjunto de **filtros** (cuyos pesos han sido aprendidos minimizando una función de coste mediante métodos basados en la retropropagación del error) son aplicados sobre un tensor de entrada para generar un

nuevo tensor de salida. Formalmente, dado un filtro K de dimensión (f, f, c) y un tensor de entrada A de dimensión (h, w, c) la operación de convolución (denotada de aquí en adelante por $\circledast$) puede definirse como [18]:

$$conv(A, K)_{x,y} = \sum_{i=1}^{f}\sum_{j=1}^{f}\sum_{k=1}^{c} K_{i,j,k}A_{x+i-1,y+j-1,k} \tag{3.1}$$

El tensor resultante será de rango dos y tendrá dimensiones:

$$dim(conv(A, K)) = (\left\lfloor \frac{h + 2p - f}{s} + 1) \right\rfloor, \left\lfloor \frac{w + 2p - f}{s} + 1) \right\rfloor) \tag{3.2}$$

donde s (*stride*) hace referencia al paso entre desplazamientos del filtro, y p (*padding*) al relleno añadido en la anchura y altura de A. Además, **sobre un mismo tensor de entrada suelen aplicarse varios filtros** concatenando todos los resultado intermedios en un tensor de rango tres. Por lo tanto, la dimensión del volumen final será $(dim(conv(A, K)), m)$ (siendo m el número de filtros)[1]. En la figura 3.4 se ilustra esta operación.

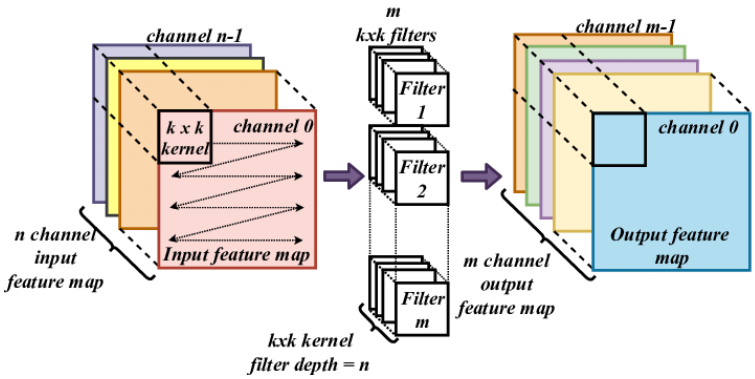


Figura 3.4: Operación de convolución aplicando m filtros sobre sobre un mapa de n canales. Imagen extraída de [55]

La convolución es una operación lineal, por lo que una sucesión de convoluciones sigue siendo una operación lineal. Para conseguir redes con capacidad para adaptarse a patrones complejos es necesario añadir, después de cada convolución, una **función de activación no lineal**. En ResNet-50 se emplea **ReLu** [19] (*Rectified Linear Unit*) como función de activación:

$$g(x) = max(0, x) = \begin{Bmatrix} x, x > 0 \\ 0, x \leq 0 \end{Bmatrix} \tag{3.3}$$

[1]Ecuación 3.2 añadiendo m como tercera componente del vector.

En resumen, es posible definir la salida (Z) de una capa convolucional como:

$$Z = g(A \circledast K + b) \tag{3.4}$$

siendo b un parámetro de sesgo.

En las capas de ***pooling*** (figura 3.5) se reduce la dimensionalidad de los mapas de características integrando valores contiguos y maximizando la información semántica relevante. Para ello se toma un filtro de tamaño (f, f) sin ningún peso entrenable para que, en su desplazamiento por el mapa, aplique una función de resumen δ (como la media o el máximo):

$$pool(A)_{x,y,z} = \delta(A_{x+i-1,y+j-1,z})_{i,j=1}^{f} \tag{3.5}$$

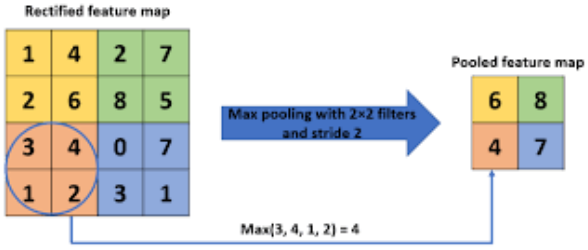


Figura 3.5: Agrupamiento por máximo con filtro de 2×2 y *stride* 2. Imagen extraída de [20]

Uno de los problemas de entrenar arquitecturas profundas es el **desvanecimiento del gradiente** [21] en la **retropropagación** del error. Durante este proceso cada peso recibe (en cada iteración) una actualización proporcional a la derivada parcial de la función de error con respecto a sí mismo. Para ello se empiezan calculando las derivadas parciales de la función de coste con respecto a los pesos de la última capa para, posteriormente, propagar este resultado hacia las capas anteriores. Es sabido que a medida que se hace esta propagación hacia atrás del gradiente, este se va **haciendo cada vez más pequeño**. Como consecuencia los pesos de las capas iniciales apenas sufren cambios en su actualización, dificultando la convergencia del algoritmo y saturando la precisión.

ResNet-50 atenúa este problema mediante el uso de **atajos** (en inglés *skip connections*). Si nos fijamos en la figura 3.3 podemos observar la presencia de estos elementos cada 3 capas convolucionales. El objetivo es permitir el aprendizaje de una **función de identidad** que garantice que las capas superiores funcionen, al menos tan bien, como las inferiores (además de permitir el flujo del gradiente a través del atajo) [22]. En la figura 3.6 se define esta operación.

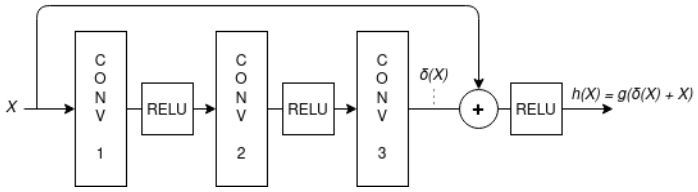


Figura 3.6: Bloque residual de tres convoluciones. g() es la función RELU. $\delta(X)$ y X deben tener la misma dimensión para que puedan ser sumados.

3.2.2. Módulo de propuestas de regiones basado en mecanismos de atención

FewX es un detector en dos etapas basado en *Faster R-CNN*. Por lo tanto, es imprescindible la existencia de algún mecanismo que se encargue de generar **regiones de interés** (RoI) sobre las cuales detectar objetos.

El módulo de propuestas de regiones se encarga, justamente, de esta tarea. Para ello no solo debe distinguir entre objetos y no objetos, sino también filtrar aquellas regiones que no contengan elementos pertenecientes a la categoría de soporte. Para conseguir esto se toma el mapa de características convolucional obtenido de procesar el conjunto de imágenes de soporte ($K_{s,s,c}$) y, mediante una operación de **correlación cruzada** (ecuación 3.6), se combina con el mapa de características de la imagen de consulta ($A_{h,w,c}$)[2]:

$$G_{h,w,c} = \sum_{i=1}^{s} \sum_{j=1}^{s} K_{i,j,c} A_{h+i-1,w+j-1,c} \tag{3.6}$$

A continuación, esta información de similitud es suministrada a un RPN (figura 3.7), encargada de generar las regiones de interés. Para ello la RPN toma el mapa $G_{h,w,c}$ y utiliza una ventana deslizante sobre él. Para cada ventana, genera K ***anchor boxes***[3] (cajas fijas y definidas como hiperparámetros) de diferentes tamaños y relaciones de aspecto[4]. Para cada *anchor* la red realiza:

- Una **clasificación** para determinar la probabilidad de que haya (o no) un objeto en la región.
- Una **regresión** para establecer las coordenadas del centro, ancho y alto de la región propuesta.

[2] (s, s, c) son las dimensiones del tensor K y (h, w, c), las dimensiones del tensor A.

[3] Se encargan de proporcionar un conjunto predefinido de cuadros delimitadores de diferentes tamaños y proporciones que se utilizan como referencia cuando se predice la ubicación de los objetos.

[4] En el artículo original [25] se toman 3 escalados y 3 relaciones de aspecto haciendo un total de 9 *anchors*.

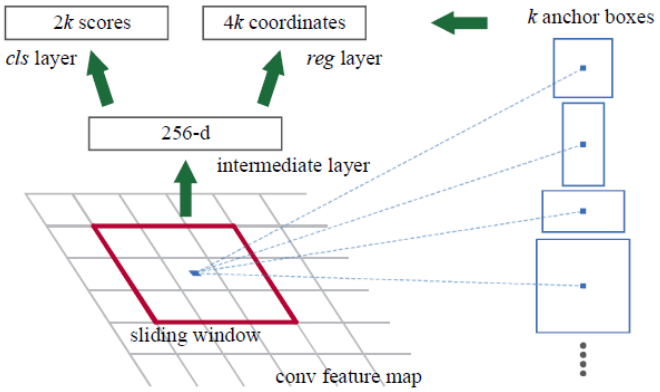


Figura 3.7: *Region Proposal Network*. Imagen extraída de [25]

Obtenidas las regiones de interés estas son redimensionadas (*RoI pooling*) y suministradas al detector de relaciones múltiple para predecir la categoría del objeto y su cuadro delimitador.

Al igual que en otras arquitecturas, entrenar una RPN requiere minimizar una función de coste a través de algún mecanismo basado en el descenso de gradiente y retropropagación del error. Para ello es necesario definir una función de coste que estime la diferencia entre los valores predichos por la red y los valores reales. En [25] se define la **función de coste** empleada en esta arquitectura como:

$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i,p_i^*) + \lambda\frac{1}{N_{reg}}\sum_i p_i^* L_{reg}(t_i,t_i^*) \tag{3.7}$$

donde $\boldsymbol{p_i}$ es la probabilidad predicha de que el *anchor* i sea un objeto; $\boldsymbol{p_i^*}$ es la etiqueta real (1 si el anchor i tiene objeto, 0 en caso contrario); $\boldsymbol{t_i}$ es el vector con las 4 coordenadas del cuadro delimitador predicho; $\boldsymbol{t_i^*}$ son las 4 coordenadas reales y $\boldsymbol{N_{cls}}$ $\boldsymbol{N_{box}}$ son términos de normalización asociados al tamaño del *batch*[5] y al número de *anchors* respectivamente.

L_{cls} es la componente de la función de coste que determina el **error en la clasificación**. En este caso se emplea una **entropía cruzada** definida como:

$$L_{cls}(p_i,p_i^*) = -p_i^* log(p_i) \tag{3.8}$$

L_{reg} es el término que determina el **error en la regresión** (predicción del cuadro delimitador). Para ello se emplea la norma L1 suavizada de la diferencia entre el cuadro real y el predicho:

$$L_{reg}(t_i,t_i^*) = L_1^{smooth}(t_i - t_i^*) \tag{3.9}$$

[5]Número de imágenes propagadas por la red antes de realizar la actualización de parámetros.

$$L_1^{smooth}(x) = \begin{cases} 0{,}5x^2 & si\ |x| < 1 \\ |x| - 0{,}5 & si\ |x| \geq 1 \end{cases} \tag{3.10}$$

Por último, en la ecuación 3.7 λ se emplea como **factor de balanceo** para equilibrar ambos términos de la suma. De esta forma los dos pasan a ser del mismo orden garantizando que ninguno de ellos influye más que el otro.

3.2.3. Detector de relaciones múltiple

Para cada una de las regiones propuestas, el detector de relaciones múltiple establece el valor de similitud respecto al soporte y ajusta el mapa de características relacionado con cada propuesta. Para lograr un mayor nivel de precisión, esta relación se realiza en tres niveles complementarios:

- **Global:** Agrupación por media de la región propuesta y el soporte por separado. Se concatenan[6] y el resultado se pasa por 2 capas completamente conectadas con activación ReLu.
- **Local:** Convolución 1x1 de la región propuesta y el soporte por separado, seguido de una correlación con activación ReLu.
- **Patch**. Se concatena la región propuesta y el soporte. El resultado se pasa por una convolución 5x5, una agrupación por media, 2 convoluciones 3x3 y otra agrupación por media.

En caso de que el soporte estuviese formado por imágenes pertenecientes a diferentes categorías, cada una de ellas añadiría a la red una nueva rama (con su propio módulo de atención RPN y detector de relaciones múltiple).

3.3. Mejoras sobre FewX

Todos los cambios, mencionados a continuación, fueron identificados e implementados iterativamente como resultado del análisis de los datos que se han obteniendo durante el estudio experimental. Los incluimos en este apartado a modo de síntesis indicando, en cada caso, su motivo y justificación.

3.3.1. Escalabilidad de la red para cualquier número de clases

El consumo de memoria de la red (en test) **crece con el número de clases**. Cuando este es suficientemente elevado **las exigencias de memoria de GPU**

[6]Para concatenar es necesario extender el soporte a la dimensión de la región propuesta.

exceden los recursos disponibles y la ejecución se detiene, haciendo imposible la experimentación con un número elevado de clases. Durante la realización de las evaluaciones, este fue uno de los problemas a los que nos tuvimos que enfrentar.

Una evaluación consta de dos fases. En la primera fase la red computa, para cada clase, los mapas de características de la información de soporte y almacena la media de los resultados en disco. En la segunda fase, carga en memoria de GPU todos estos mapas precomputados y los relaciona, uno a uno,[7] con la imagen de consulta.

Cuanto mayor es el número de clases, mayor es el número de mapas de soporte precomputados y, consecuentemente, mayor es la cantidad de espacio de GPU necesario para almacenarlos. En el límite, el tamaño de los mapas de soporte es superior a la cantidad de memoria de GPU disponible y el proceso termina con el error *cuda out of memory*.

Para solucionar este problema (y conseguir una red escalable para cualquier número de clases) lo que se ha hecho es **discretizar la carga de soportes** precomputados en GPU. Es decir, en vez de cargar simultáneamente los soportes de todas las clases en memoria, lo que se hace es cargar n_{clases} por iteración. En la primera iteración se obtienen las predicciones de los objetos de las n primeras clases, en la segunda, de las n siguientes y así sucesivamente. Finalmente, se concatenan todos los resultados.

n_{clases} es un parámetro que puede ser fijado por el usuario o estimado automáticamente a partir del espacio disponible en GPU, el tamaño del modelo, y el tamaño de los mapas de soporte.

Finalmente, para acelerar el procedimiento y evitar desplazamientos innecesarios de datos entre disco y GPU, lo que se hace es cargar al inicio todos los soportes en memoria principal. Así conseguimos que los únicos movimientos se den entre memoria y GPU, que es varias órdenes de magnitud más rápido que entre disco y GPU.

Como resultado de estos cambios, las evaluaciones son escalables para cualquier número de clases y son un 26 % más rápidas.

[7] En total, el número de veces que se ejecuta un ciclo completo de red es el producto del número de clases por el número de imágenes de consulta.

3.3.2. Escalabilidad de la red para cualquier tamaño de shot

Durante el entrenamiento de los modelos, se observa que el **consumo de memoria de la red crece con el número de imágenes de soporte por clase** (*k-shot*). Al realizar pruebas con *k-shot* grandes (del orden de 100 y superiores) nos dimos cuenta de que las exigencias de memoria GPU exceden los recursos disponibles y el proceso termina con el error *cuda out of memory*. Esto imposibilitaba algunos de los casos de prueba diseñados, limitando el estudio a valores de *k-shot* pequeños y dejando fuera del análisis el comportamiento de la red para tamaños altos del conjunto de soporte. Para solucionar este problema se ha modificado la forma en la que se hace la propagación hacia adelante de las imágenes.

La versión original de FewX converge a los pesos óptimos minimizando una función de coste a través de un algoritmo de **descenso de gradiente por lotes** (comúnmente denominados *batch*) de tamaño 8. Es decir, la actualización de los parámetros se realiza con el error obtenido cada vez que 8 propagaciones hacia adelante son computadas.

Cada uno de estos 8 bloques está integrado por 1 imagen de consulta, *k-shot* imágenes de soporte positivas y *k-shot* imágenes de soporte negativas (estrategia de entrenamiento por contraste mencionada en el apartado 3.2)[8].

La implementación original de FewX **no es escalable** porque en la extracción de características de la rama de soporte, se generan y almacenan en GPU $|batch| \times (2 \times k_{shot})$ mapas para, posteriormente, calcular 2 mapas promedios (uno de la clase positiva y otro de la clase negativa). Es decir, a pesar de que el objetivo final es obtener únicamente 2 mapas de características que resuman toda la información de soporte de sus respectivas clases, la red almacena en GPU todos los mapas empleados para hacer este cálculo. Lógicamente, cuando *k-shot* es suficientemente elevado el programa falla por escasez de memoria GPU.

Para solucionar este problema, en vez de procesar las características de todas las imágenes de soporte simultáneamente, se hace en **intervalos discretos de** e **elementos** [9]. Se suman todos los e mapas intermedios sobre una única variable y, finalmente, se calcula la media. Este procedimiento es aplicado tanto con la clase positiva, como con la negativa.

[8] En total, el *batch* contiene $8 \times (2 \times k_{shot} + 1)$ imágenes.
[9] El último intervalo puede tener menos de e elementos

3.3.3. *Feature Pyramid Network* (FPN)

Analizando los resultados experimentales, se observó que el desempeño de la red FewX para objetos pequeños era considerablemente bajo. Como solución se optó por introducir una FPN.

Una *Feature Pyramid Network* [32] es un extractor de características específicamente diseñado para detectar objetos en diferentes escalas. Para ello toma como entrada una imagen de tamaño arbitrario y produce mapas de características convolucionales de tamaño proporcional en múltiples niveles (figura 3.8).

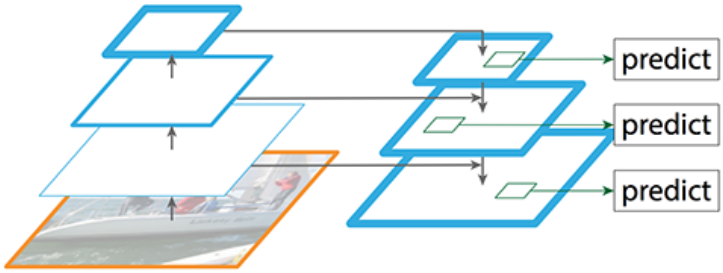


Figura 3.8: FPN. Imagen extraída de [32]

La construcción de la pirámide implica la necesidad de dos vías, una **ascendente** y otra **descendente**. La vía ascendente es una red convolucional tradicional (en nuestro caso ResNet). A medida que se sube, la resolución espacial se reduce, pero la información de cada capa aumenta. La vía descendente tiene por objetivo enriquecer la información semántica de las capas más bajas (de mayor resolución) para facilitar en ellas la detección de objetos pequeños. Con tal fin, los mapas de características de mayor nivel son relacionados con los mapas de las capas de la vía ascendente a través de conexiones laterales.

En la figura 3.9 podemos observar gráficamente ambas operaciones aquí explicadas:

- **Vía ascendente**. Un conjunto de módulos convolucionales (conv1, conv2, ..., conv5) son aplicados a la imagen de entrada para generar 5 mapas de características intermedios en diferentes escalas (C1, C2,...,C5).
- **Vía descendente**. Se genera el conjunto final de mapas de características multinivel (P2, P3, ..., P5). M5, surge de aplicar 256 convoluciones 1x1 a C5 y pasa a ser directamente P5. Los restantes Pi se obtienen de aplicar una convolución a la suma elemento a elemento del mapa superior (extendido a la dimensión del actual) más el mapa lateral (reducido a 256 canales).

Es importante destacar que el número de mapas, M_i y P_i, puede variar entre implementaciones.

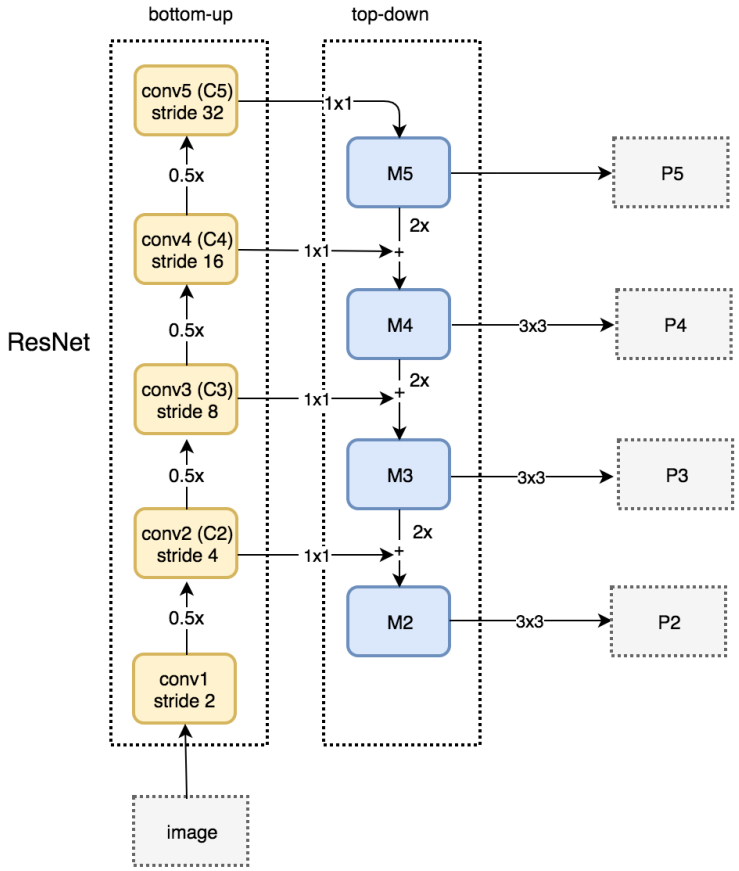


Figura 3.9: A la izquierda vía ascendente. A la derecha vía descendente. Imagen extraída de [32]

Para incluir correctamente la FPN en la red FewX es necesario realizar una serie de modificaciones adicionales:

- **Modificar la red de pesos compartidos**. El extractor FPN debe emplearse tanto para procesar la imagen de consulta, como las *k-shot* imágenes de soporte.

- **Modificar la RPN basada en mecanismos de atención**. Ahora la correlación entre el soporte medio y la consulta debe realizarse en todas las escalas P1, P2, ..., P5. Es decir el mapa P1 de soporte se correlaciona con el mapa P1 de consulta para generar un nuevo mapa (más preciso) sobre el cual generar regiones de interés.

- **Modificar la cabecera de la red**[10]. En la versión original se empleaba un bloque res-5. Ahora se emplean 2 capas convolucionales 3x3.

La definición detallada de la arquitectura puede consultarse en el apéndice B.

3.4. Materiales

3.4.1. Servidores de GPUs

El entrenamiento y evaluación de redes neuronales convolucionales requiere de una extraordinaria capacidad de cómputo para converger a los resultados en un tiempo razonable. Una forma de lograrlo es explotar el paralelismo a nivel de datos empleando GPUs.

Por ello, hemos tenido acceso a un conjunto de servidores de computación con las siguientes características:

- Servidor **Dell PowerEdge R740** (ver especificaciones en [43]).
- 2 procesadores **Intel Xeon Gold 5220** (ver especificaciones en [41]).
- **192 GB** de memoria RAM (12 DDR4 DIMM a 2667MHz).
- 2 **Nvidia Tesla V100S** [42]. Cada una permite hasta 130 teraFLOPS, 32 GB de memoria HBM2 y 1134 GB/s de ancho de banda.
- Sistema operativo **CentOS** 8.1.
- Driver Nvidia 440.64.00 para **CUDA** 10.2 y **Docker** 19.03.

Mencionar que todas las tareas con necesidades de computación intensivas (principalmente entrenamientos y evaluaciones) son realizadas remotamente sobre estos equipos.

3.4.2. Lenguajes de programación

Para el desarrollo del proyecto se ha optado por:

- **Python** [44]. Es un lenguaje de programación interpretado, multiparadigma y de alto nivel. Cuenta con una popularidad creciente y es ampliamente utilizado para la construcción de sistemas de Inteligencia Artificial. Nos hemos decantado por él, puesto que la versión de la red de la que partimos

[10]Entendida como el conjunto de capas que siguen al *RoI Pooling* y preceden a las capas finales de clasificación y regresión.

fue desarrollada utilizando este lenguaje. Por consiguiente, todas las modificaciones a elaborar tienen que ser realizadas en Python. Además, cuenta con un diverso ecosistema de librerías que facilitan el desarrollo de este tipo de aplicaciones. Algunas de las que empleamos son:

  - **Pytorch** [47]. Especializada en el desarrollo de sistemas de aprendizaje automático. Incluye todas las operaciones básicas necesarias para elaborar implementar neuronales.
  - **Pandas**. Facilita el manejo y manipulación de información estructurada. Utilizada para gestionar el conjunto de datos.
  - **Scikit-image, pillow y openCV**. Para manipulación y procesamiento de imágenes.
  - **NumPy**. Soporte para vectores y matrices multidimensionales.

- **Bash**. Es un lenguaje de órdenes, interpretable por sistemas UNIX. Se utiliza para automatizar los experimentos y transformar los resultados a un formato explotable. El principal motivo de su elección radica en que nos permite elaborar *scripts* directamente ejecutables en el servidor (el servidor dispone de SO CentOS con consola bash).

3.4.3. Detectron2

La red *FewX* está construida sobre Detectron2 [46], una biblioteca de código abierto (desarrollada por Facebook) que dispone de un conjunto de **utilidades para la detección de objetos**.

Detectron2 emplea el lenguaje interpretado Python en combinación con el *framework* de aprendizaje profundo PyTorch. Funciona sobre sistemas GNU/Linux y sus algoritmos pueden ejecutarse fácilmente en GPUs a través de CUDA. Esto último mejora el rendimiento y reduce el tiempo de ejecución de los entrenamientos e inferencia de los modelos.

3.4.4. CUDA

CUDA [48] es una plataforma de **computación en paralelo** y un modelo de programación desarrollado por NVIDIA para el cálculo general en unidades de procesamiento gráfico (GPGPU). Con CUDA, pueden ejecutarse los entrenamientos/evaluaciones más rápido, aprovechando la potencia de las **GPUs**.

3.4.5. Docker

Docker [45] es una herramienta de gestión de contenedores. Es empleada para desplegar los ejecutables en el servidor.

Docker utiliza un fichero denominado *Dockerfile*, que incluye toda la información necesaria para crear un entorno en el que se pueda ejecutar la aplicación. Este fichero es empleado para construir una imagen, es decir, una plantilla compilada equipada con todo lo necesario para que la aplicación pueda funcionar (binarios, dependencias y librerías). A continuación, la imagen es instanciada en un contenedor.

En nuestro caso particular partimos de un *Dockerfile* que genera una imagen funcional de *detectron2* sobre la cual instalamos *FewX* y todas sus dependencias. Cuando queremos ejecutar la red en el servidor, instanciamos un contenedor a partir de esta imagen y ejecutamos en él la aplicación.

Como alternativa, también se podría haber utilizado entornos virtuales de Anaconda para Python. Sin embargo, optamos por la primera opción, puesto que Docker ya se encuentra instalado en los servidores.

3.4.6. Herramientas

Se ha empleado:

- **Visual Studio Code** para la codificación [49].
- **Overleaf** para la redacción de esta memoria [50].
- **Diagrams.net** para la creación de diagramas [51].
- **Google Sheets** para recoger, analizar e interpretar los datos extraídos de los experimentos.

Capítulo 4

Pruebas

4.1. Diseño experimental

4.1.1. *Conjunto de datos*

Un conjunto de datos es una colección de información estructurada, comúnmente utilizada para entrenar y evaluar algoritmos de aprendizaje automático. En nuestro caso particular (detección) serán **imágenes** etiquetadas con, al menos, la **ubicación** y **clase** de los objetos a detectar.

Existen muchos conjuntos de datos válidos para el entrenamiento y evaluación de detectores. Entre algunos de los más importantes se encuentran MS COCO [36], ImageNet DET [37], CIFAR-10 [38] o PASCAL VOC [39].

En este trabajo se empleará únicamente **MS COCO** (*Microsoft Common Objects in Context*). Tres son los motivos de está decisión. En primer lugar, es un *dataset* suficientemente amplio (>200 000 imágenes) para los objetivos a alcanzar. Además, es de acceso libre pudiendo ser descargado gratuitamente desde su página web ([40]). Por último, la mayor parte de los trabajos en el ámbito de la detección de objetos mediante *few-shot* (incluido [1]) publican sus resultados sobre esta biblioteca de imágenes.

MS COCO

La primera versión de MS COCO fue lanzada en 2014. Contenía 164 000 imágenes divididas en dos particiones: una de entrenamiento con 83 000 imágenes y otra de validación con 41 000.

En este trabajo se utiliza la **versión de 2017**. Utiliza las mismas **164 000 imágenes** que su antecesora, pero modificando el número de elementos que conforman cada partición (de 83 000/41 000 a **118 000/5 000**). Adicionalmente, se

añaden 123 000 imágenes no etiquetadas, útiles para aprendizaje no supervisado, pero que en nuestro caso no serán empleadas.

Al tratarse de un *dataset* apto para la tarea de detección, cada imagen está anotada con las instancias de los objetos a detectar. De esta forma, el número total de objetos es 860 001 para el conjunto de entrenamiento y 36 781 para el de validación. Cada uno de estos objetos está asignado a una de las **80 clases categorizadas**. La distribución específica de objetos por clase puede consultarse en el apéndice A.

Para evaluar el desempeño de un algoritmo *few-shot* es necesario que las **clases de validación (C_{nuevas}) sean diferentes a las clases de entrenamiento (C_{base})**. Por consiguiente, la división actual del *dataset* **no es válida**, ya que las 80 categorías se repiten en ambos conjuntos. Para solucionar este problema se han adoptado las configuraciones de trabajos previos ([1], [4], [28] y [29] entre otros):

- **Separación de las clases en entrenamiento/test:** Se toman las categorías 1-person, 2-bicycle, 3-car, 4-motorcycle, 5-airplane, 6-bus, 7-train, 9-boat, 16-bird, 17-cat, 18-dog, 19-horse, 20-sheep, 21-cow, 44-bottle, 62-chair, 63-couch, 64-potted plant, 67-dining table y 72-tv como C_{nuevas} (20 en total) y, las restantes, como C_{base} (60 en total)[1].
- **Subconjunto de entrenamiento:** Se entrenan los modelos sobre un subconjunto de la partición de entrenamiento de COCO conformado, exclusivamente, por las 60 clases base (D_{base}). La información de soporte es extraída del propio D_{base}, seleccionando *N-shot* instancias por categoría.
- **Subconjunto de test:** Se evalúan los modelos sobre un subconjunto de la partición de validación de COCO conformado, únicamente, por las 20 clases nuevas (D_{nuevo}). En este caso la información de soporte es extraída del conjunto de entrenamiento (únicamente de las 20 clases nuevas que no se emplearon durante el proceso).

Por último, mencionar que las imágenes que integran este *dataset* no solo están etiquetadas para realizar detección de objetos; sino que también incluyen información útil para llevar a cabo tareas de segmentación, estimación de la posición humana, etc. Nosotros prescindiremos de esta información y nos centraremos, exclusivamente, en las anotaciones necesarias para lograr los objetivos de detección.

Estas anotaciones se encuentran en un archivo JSON con el siguiente formato (todos los campos son autoexplicativos):

[1]Las 20 clases nuevas se solapan con las presentes en PASCAL VOC para facilitar la interoperación entre conjunto de datos.

```
'annotations': [
    {
        'segmentation': [[510.66,...,423.01]],  #ignorado
        'area': 702.1057499999998,
        'iscrowd': 0,  #ignorado (se usa en segmentacion)
        'image_id': 289343,
        'bbox': [473.07,395.93,38.65,28.67],
        'category_id': 18,
        'id': 1768
    },
    ...
    {...}
]
```

4.1.2. Métrica de evaluación

La métrica de evaluación que se va emplear para medir el desempeño de los modelos será el **mAP** (*Mean Average Precision*) sobre diferentes umbrales de **IoU** (intersección sobre unión). El motivo radica en que es la métrica estándar en el campo de la detección de objetos. ([31]).

El IoU (figura 4.1) determina la concordancia entre el cuadro delimitador predicho y el cuadro real. Se calcula como el cociente de la intersección de ambos entre su unión. Si ambas áreas son iguales el resultado es 1. Si no tienen nada en común, 0.

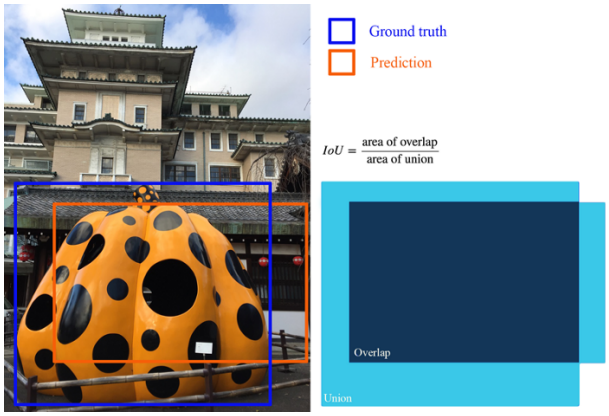


Figura 4.1: Intersección sobre unión. Imagen extraída de [56]

El umbral es un valor preestablecido que determina el IoU mínimo necesario para considerar correcta una predicción. Es decir, si $IoU > umbral$ la predicción

se considera correcta (verdadero positivo, VP). Si $IoU < umbral$ la predicción se considera incorrecta (falso positivo, FP).

Definidos los umbrales, la precisión y el *recall* se calculan como:

$$Precision = \frac{VP}{VP + FP} \tag{4.1}$$

$$Recall = \frac{VP}{VP + FN} \tag{4.2}$$

donde FN es el número de objetos reales no detectados.

La precisión determina, de entre todas las predicciones positivas, cuantas son realmente positivas. El *recall* determina, de entre todos los objetos en la imagen, cuantos son correctamente detectados. El AP se define como el área bajo la curva *recall*-precisión.

Una forma frecuente de calcular esta área es interpolar los resultados para diferentes valores de *recall* y calcular la media del sumatorio de la precisión en cada uno de ellos (figura 4.2).

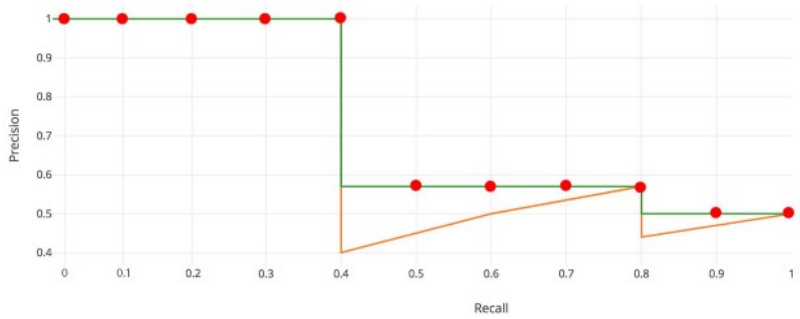


Figura 4.2: Cálculo del AP interpolando la función sobre 11 puntos de *recall*. $AP = \frac{1}{11} \sum_{r \epsilon \{0,0,\ldots,1,1\}} p_{interp}(r)$. Imagen extraída de [56]

Finalmente, es importante destacar que nuestros resultados serán reportados siguiendo el estándar definido en COCO (con interpolación de 101 puntos) donde:

- **AP**: media de AP con IoU de 0.5 a 0.95 con paso de 0.005. Es la métrica principal.
- **AP50**: AP con IoU de 0.5.
- **AP75**: AP con IoU de 0.75.
- **APs**: AP para objetos con *área* $\leq 32^2$

- **APm**: AP para objetos con $32^2 < \textit{área} < 96^2$
- **APl**: AP para objetos con $\textit{área} \geq 96^2$

Calculado el AP para una clase, el mAP es la media de los AP de todas las clases.

4.1.3. Flujo de generación de datos para test

Para facilitar la automatización de los experimentos, se ha modificado la secuencia de pasos necesarios para obtener los datos de evaluación (imágenes de consulta, soporte y anotaciones). El proceso es el siguiente (figura 4.3).

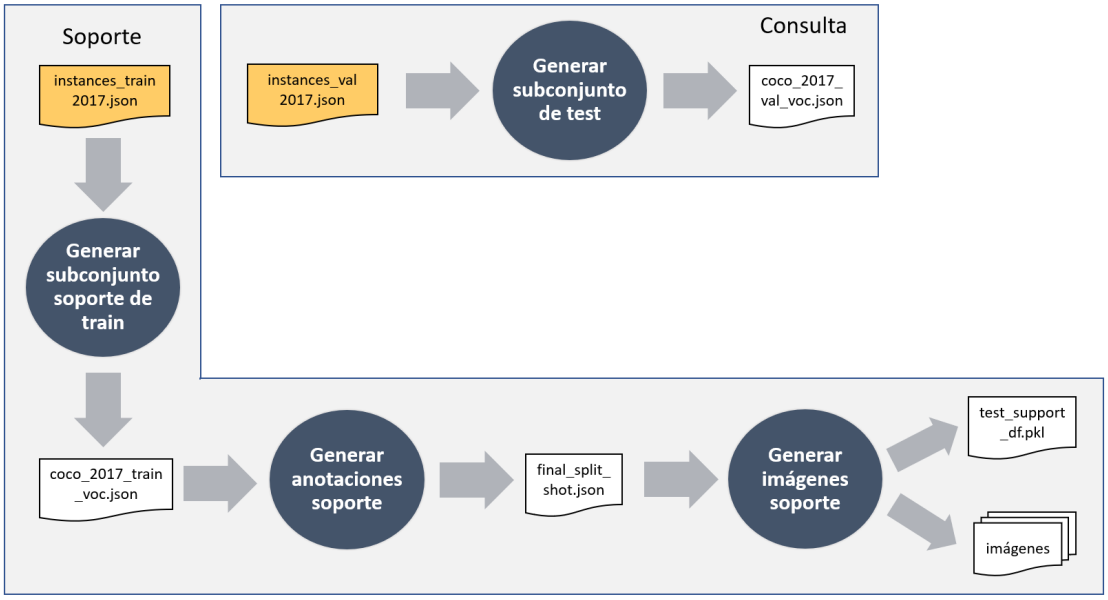


Figura 4.3: Flujo de generación de datos de test.

Se observa la existencia de dos ramas principales. La superior se encarga de filtrar las imágenes de consulta y prepararlas para su introducción en la red. Para ello cuenta con un único proceso [2]:

- **Generar subconjunto de test**. Partiendo del archivo de anotaciones de la partición de validación de COCO, se genera un subconjunto de datos integrado exclusivamente por imágenes con objetos pertenecientes a una de las 20 clases nuevas (C_{nuevas}). Estas serán empleadas como imágenes de consulta.

La rama inferior genera la información de soporte. Esta compuesta por tres procesos:

[2]Tiene asociado el script generar_voc_annotations.py

- **Generar subconjunto soporte de train**. Partiendo del archivo de anotaciones de la partición de entrenamiento de COCO, se genera un subconjunto de datos integrado únicamente por C_{nuevas}.

- **Generar anotaciones de soporte**. Se toman *k-shot* imágenes aleatorias del subconjunto previamente creado y se almacenan sus anotaciones en un nuevo fichero en formato JSON. Para hacer los experimentos replicables y extensibles a un amplio abanico de casos de prueba, la semilla aleatoria y el número de imágenes de soporte (*k-shot*) son parámetros seleccionables.

- **Generar imágenes de soporte**. Se toma el archivo JSON anterior y se recorta, en cada imagen, los cuadros delimitadores de los objetos a detectar. A continuación, estos recortes son escalados a dimensión 320x320 y almacenados. Los recortes más sus anotaciones constituyen la información de soporte.

Adicionalmente, es necesario registrar en detectron2 los nuevos *datasets* (ficheros de anotaciones JSON) para que sean directamente accesibles desde el archivo de configuración de la red. También se debe modificar, en el código, la ruta a los directorios de imágenes.

Por último, destacar que el flujo de datos de entrenamiento permanece inalterado, es decir, los cambios mencionados solo afectan a la generación de los datos de la parte de test.

4.1.4. Detalles de la experimentación

- El término *k-shot* indica el número de imágenes de soporte (por clase) empleadas durante el test.

- El término *F-shot* indica el número de imágenes (por clase) con las que se hace *finetuning*.

- El término *X-shot_base* hace referencia a un modelo entrenado únicamente con soporte de X imágenes por clase base (C_{base}).

- El término *X-shot_fine-F* hace referencia a un modelo base, *X-shot*, reentrenado con la información de soporte de test, es decir rentrenado con *F-shot* imágenes de soporte de las clases nuevas (C_{nuevas}).

- El sufijo FPN (*X-shot_base_FPN* o *X-shot_fine-F_FPN*) es análogo a los dos anteriores, pero indicando que se ha empleado la versión de la red con FPN.

- Para mitigar el efecto de la variabilidad entre soportes, y salvo que se diga lo contrario, los experimentos son promediados sobre 3 conjuntos de soporte aleatorios distintos (pero siempre los mismos entre experimentos).
- Los entrenamientos de la versión de *FewX* sin FPN son realizados con la configuración preestablecida:
  - Entrenamiento base: 8 de tamaño de *batch*, 120 000 iteraciones, 0.004 de *learning rate* con reducción a 0.0004 en la iteración 112 000 y pesos preentrenados en *ImagiNet*.
  - Entrenamiento de *finetuning*: 4 de tamaño de *batch*, 3 000 iteraciones y 0.001 de *learning rate* con reducción a 0.0001 en la iteración 2000.
- Las pruebas que requieran el uso de un único modelo utilizarán *10-shot_fine-10*. Esta es la configuración que suele emplearse para comunicar los resultados en otros trabajos del estado del arte [4], [1], [29] (junto con *3-shot_fine-3* o *30-shot_fine-30*).
- Experimentalmente estudiaremos cómo influye en el desempeño del modelo *"FewX"*:
  1. El grado de **similitud entre** C_{base} **y** C_{nuevas}. Es decir, evaluar como varía el rendimiento del modelo al realizar el test sobre categorías más o menos parecidas a las de entrenamiento.
  2. La variabilidad de resultados entre soportes. Determinar la influencia en la selección de unas imágenes u otras.
  3. La influencia del tamaño del soporte (*k-shot*) en el AP final.
  4. La influencia del tamaño de *F-shot* en el AP final.
  5. La posible mejora que introduce la implementación de una ***Feature Pyramid Network*** (FPN) para la detección de objetos de diferentes tamaños.

4.2. Experimento 1: Variación entre soportes

El objetivo es analizar la influencia de la variabilidad del conjunto de soporte en el desempeño de la red. Es decir, determinar cuanto varían los resultados al probar con soportes distintos del mismo tamaño.

Este experimento puede dividirse en dos partes. En la primera se toma el modelo *10-shot_base*[3] y se evalúa su desempeño sobre 40 soportes disjuntos (sin imágenes en común). La tabla 4.1 recoge la desviación y media muestrales de los resultados para las distintas métricas consideradas.

Métrica	$\overline{x}$	σ
AP	5.62	0.44
AP50	10.60	0.71
AP75	5.39	0.50
APs	2.70	0.29
APm	5.80	0.60
APl	8.02	0.70

Tabla 4.1: Resultados experimentales para *10-shot_base* sobre 40 soportes disjuntos. Se incluye la desviación y media muestrales.

Los resultados más significativos de esta tabla son las desviaciones. Particularmente un σ de 0.44 para AP (métrica principal) cuantifica la variabilidad entre soportes. Es decir, indica que realizar las evaluaciones sobre conjuntos de soporte distintos implica que los resultados obtenidos puedan desviarse en promedio 0,44 puntos.

Entre métricas los resultados mantienen un comportamiento proporcional: cuanto mayor es $\overline{x}$, más ancho es el potencial rango de posibles valores y, consecuentemente, mayor es la desviación. Por ejemplo, σ AP50 (0.71) es aproximadamente el doble que σ AP (0.44), pero las medias también lo son (10.60 frente 5.62).

En la segunda parte, lo que interesa es determinar la variación de soportes entre modelos. Para ello se toma un modelo y se evalúa su rendimiento sobre tres soportes disjuntos para los siguientes valores de *k-shot* 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, y 300. Por último, obtenemos la desviación

[3] *10-shot_base* (junto a la versión con *finetuning*) es el modelo de referencia sobre el cual se suelen comunicar los resultados en este campo [4], [1], [29]. Por eso lo hemos empleado en esta primera parte del experimento.

estándar para cada *k-shot* (21 en total) y nos quedamos con el promedio. Este proceso lo repetimos con las tres familias de modelos:

- *10-shot_base* y *10-shot_base_fine-i* para $i \in \{10,\ 50, 150, 200, 300\}$
- *50-shot_base* y *50-shot_base_fine-i* para $i \in \{10,\ 50, 150, 200, 300\}$
- *150-shot_base* y *150-shot_base_fine-i* para $i \in \{10,\ 50, 150, 200, 300\}$

y nos quedamos con la media.

La tabla 4.2 recoge los resultados de esta prueba. La segunda, tercera y cuarta columnas reúnen el promedio de las desviaciones para las familias de modelos *10-shot*, *50-shot* y *150-shot*, es decir, los modelos base además de sus correspondientes reentrenamientos de *finetuning*.

	fam. 10-shot	fam. 50-shot	fam. 150-shot
Métrica	σ	σ	σ
AP	0.42	0.44	0.48
AP50	0.66	0.76	0.98
AP75	0.50	0.49	0.56
APs	0.38	0.39	0.48
APm	0.75	0.51	0.74
APl	0.72	0.87	0.92

Tabla 4.2: Desviaciones y medias muestrales para las 3 familias de modelos. Cada una está promediada sobre los 21 valores de *k-shot* y los 6 modelos.

Aunque la familia 150-shot es la que presenta una desviación mayor entre soportes, los resultados permanecen prácticamente constantes (a excepción del AP50, donde $0{,}98 > 0{,}76 > 0{,}66$ y APl, donde $0{,}92 > 0{,}87 > 0{,}72$). Además, si se comparan estos resultados con los de la tabla 4.1 podemos ver que uno y otro son significativamente parecidos. Ambos hechos ponen de manifiesto que la variabilidad entre soportes depende exclusivamente del conjunto de imágenes empleado y no del modelo considerado.

4.3. Experimento 2: Variación del AP frente a *F-shot* y *K-shot*

Dos son los objetivos de este experimento. Por un lado, evaluar como influye el tamaño del soporte (*k-shot*) en test. En segundo lugar, analizar la influencia

del número de imágenes (*F-shot*) con las que se realiza *finetuning* sobre diferentes modelos base.

Para lograrlo se entrenan los modelos *10-shot_base*, *50-shot_base* y *150-shot_base*. Para cada uno de ellos, se realiza un *finetuning* de 10, 50, 150, 200, 300 y 400[4]. Finalmente se evalúa el rendimiento de todos los 21 modelos sobre tres conjuntos de soporte diferentes para los siguientes tamaños de *k-shot*: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, y 300.

Los resultados de este experimento se recogen en las tablas 4.3, 4.4 y 4.5. La tabla 4.3 reúne los resultados de AP para el modelo *10-shot_base* más sus correspondientes entrenamientos de *finetuning*. Las tablas 4.4 y 4.5, lo mismo, pero con las familias 50-shot y 150-shot respectivamente.

Las figuras 4.4, 4.5 y 4.6, resumen esta información de forma visual. En cada una de ellas, la subfigura a) muestra la comparativa (AP) de los diferentes modelos de una familia. La subfigura b) la mejora (en tanto por 1) del AP de un modelo frente a su anterior (el punto 1 refleja la ganancia del modelo *x-shot_base* frente al *x-shot_fine-10*, el 2 del modelo *x-shot_fine-50* frente al *x-shot_fine-10* y así sucesivamente).

En general, es fácil observar que en todos los modelos **el AP es creciente hasta llegar a un umbral de *k-shot* en el cual satura**. Esta saturación depende directamente de dos factores:

1. El número de imágenes de soporte que se usaron durante en el entrenamiento base.
2. El valor de *F-shot*.

En lo referente al primer factor, en un análisis entre los modelos base se observa que cuanto mayor es el tamaño del soporte durante el entrenamiento, mayor es el AP obtenido y más tarde se llega al umbral de saturación. Concretamente, *10-shot_base* se satura en torno a 30 shots con un AP de 6.19, *50-shot_base* en torno a 100 shots con un AP de 7.73 y *150-shot_base* en torno a 200 shots con un AP de 8.49.

En lo que respecta al *finetuning*, si se mantiene fija una familia de modelos (por ejemplo la de 10-shot) se observa que el modelo base (sin *finetuning*) satura en torno a 30 shots con un AP de 6.19, mientras que *10-shot_base_fine-300*, sobre 200 con un AP de 14.07. Es decir, cuanto mayor es el tamaño del soporte empleado en el *finetuning*, mayor es el AP y más tarde se alcanza el umbral de saturación.

[4]Para *10_shot_base*, no realizamos el *finetuning* con 400, puesto que con 300 ya converge.

Además, también se aprecia que la mejora que supone el uso de *finetuning* disminuye al aumentar *F-shot*. En la figura 4.4 puede observarse que *10-shot_fine-50* es un 40 % mejor que *10-shot_fine-10*; *10-shot_fine-150*, un 10 % mejor que *10-shot_fine-50* y así sucesivamente hasta que la ganancia pasa a ser negativa (*10-shot_fine-300* es un 0.1 % peor que *10-shot_fine-200*). Este comportamiento se repite para las familias 50-shot (figura 4.5) y 150-shot (figura 4.6), donde las ganancias entre los modelos *fine-300* y *fine-400* son 1.1 % y 0.4 % respectivamente.

	k-shot																				
fam. 10-shot	1	2	3	4	5	6	7	8	9	10	20	30	40	50	60	70	80	90	100	200	300
base	3.24	4.37	5.09	5.58	5.58	5.47	5.49	5.62	5.68	5.76	5.93	6.19	6.19	6.19	6.27	6.37	6.36	6.39	6.41	6.36	6.31
fine_10	5.05	6.69	7.35	7.72	7.98	7.92	7.94	8.01	8.11	8.19	8.43	8.57	8.49	8.44	8.41	8.44	8.44	8.47	8.45	8.37	8.30
fine_50	6.05	8.16	9.69	10.50	10.89	10.89	10.88	11.17	11.31	11.59	12.05	12.56	12.58	12.59	12.63	12.59	12.59	12.58	12.66	12.45	12.29
fine_150	6.43	8.95	10.46	11.19	11.59	11.60	11.55	11.94	12.08	12.37	13.04	13.52	13.57	13.61	13.56	13.60	13.66	13.66	13.78	13.76	13.62
fine_200	6.77	9.29	10.97	11.62	12.07	12.19	12.23	12.46	12.62	12.89	13.49	14.07	14.06	14.12	14.20	14.24	14.30	14.35	14.42	14.50	14.37
fine_300	6.54	9.16	10.62	11.45	11.77	11.94	11.94	12.25	12.38	12.69	13.10	13.66	13.73	13.68	13.79	13.92	13.95	13.93	13.96	14.07	13.92

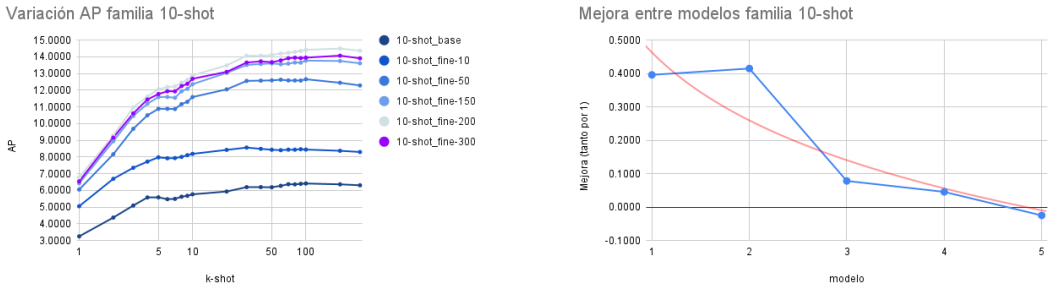
Tabla 4.3: Resultados experimentales de AP para la familia de modelos 10-shot. Se incluye el modelo base y sus *finetuning* 10, 50, 150, 200 y 300.

	k-shot																				
fam. 50-shot	1	2	3	4	5	6	7	8	9	10	20	30	40	50	60	70	80	90	100	200	300
base	3.48	4.80	5.81	6.48	6.34	6.18	6.30	6.38	6.45	6.68	7.10	7.47	7.52	7.55	7.60	7.63	7.64	7.68	7.73	7.70	7.64
fine_10	5.16	7.04	7.70	7.92	8.18	8.31	8.27	8.43	8.48	8.60	9.07	9.22	9.09	9.04	9.05	9.02	9.02	9.02	9.01	8.86	8.75
fine_50	6.22	8.57	10.01	10.86	11.18	11.25	11.23	11.54	11.63	11.94	12.53	12.88	13.01	13.05	13.03	12.99	13.00	12.93	13.01	12.80	12.71
fine_150	6.80	9.23	11.23	12.03	12.28	12.30	12.41	12.84	13.08	13.42	14.16	14.67	14.85	14.86	14.91	14.99	14.94	14.99	15.04	15.07	14.96
fine_200	6.94	9.47	11.48	12.36	12.80	12.77	12.96	13.22	13.48	13.83	14.67	15.24	15.38	15.39	15.49	15.58	15.60	15.59	15.69	15.72	15.64
fine_300	6.73	9.27	11.16	12.11	12.58	12.60	12.71	13.18	13.46	13.78	14.78	15.28	15.57	15.58	15.67	15.69	15.72	15.75	15.77	15.79	15.72
fine_400	7.00	9.51	11.40	12.34	12.76	12.84	13.02	13.26	13.53	13.91	14.93	15.45	15.63	15.58	15.69	15.70	15.70	15.84	15.90	15.97	15.86

Tabla 4.4: Resultados experimentales de AP para la familia de modelos 50-shot. Se incluye el modelo base y sus *finetuning* 10, 50, 150, 200, 300 y 400.

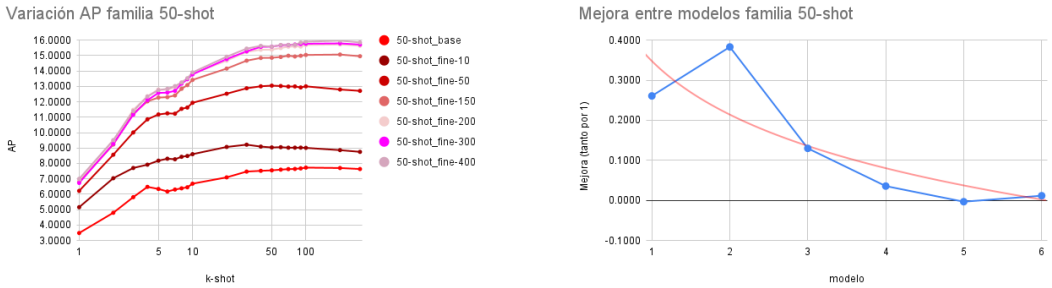
	k-shot																				
fam. 150-shot	1	2	3	4	5	6	7	8	9	10	20	30	40	50	60	70	80	90	100	200	300
base	3.25	4.59	5.83	6.71	6.82	6.86	6.94	7.11	7.20	7.38	7.66	8.20	8.20	8.27	8.28	8.40	8.35	8.45	8.49	8.49	8.51
fine_10	4.86	6.48	7.39	7.75	8.00	8.03	8.08	8.22	8.30	8.39	8.53	8.72	8.68	8.66	8.68	8.70	8.65	8.63	8.63	8.51	8.41
fine_50	5.28	7.56	9.44	10.22	10.59	10.75	10.84	10.96	11.08	11.25	11.70	12.11	12.24	12.19	12.26	12.24	12.24	12.25	12.23	12.12	12.02
fine_150	5.81	8.11	10.25	11.02	11.47	11.61	11.68	12.02	12.15	12.48	13.21	13.80	13.86	13.96	14.00	14.08	14.08	14.16	14.25	14.36	14.21
fine_200	5.93	8.40	10.52	11.25	11.73	12.02	12.12	12.55	12.85	13.14	13.73	14.24	14.37	14.41	14.41	14.48	14.52	14.58	14.69	14.76	14.73
fine_300	5.90	8.52	10.63	11.47	12.09	12.20	12.25	12.56	12.82	13.00	13.79	14.51	14.59	14.65	14.65	14.72	14.79	14.83	14.88	14.94	14.91
fine_400	6.07	8.22	10.44	11.29	11.87	12.14	12.32	12.59	12.90	13.25	14.00	14.74	14.77	14.82	14.78	14.82	14.91	14.98	15.07	15.08	15.14

Tabla 4.5: Resultados experimentales de AP para la familia de modelos 150-shot. Se incluye el modelo base y sus *finetuning* 10, 50, 150, 200, 300 y 400.



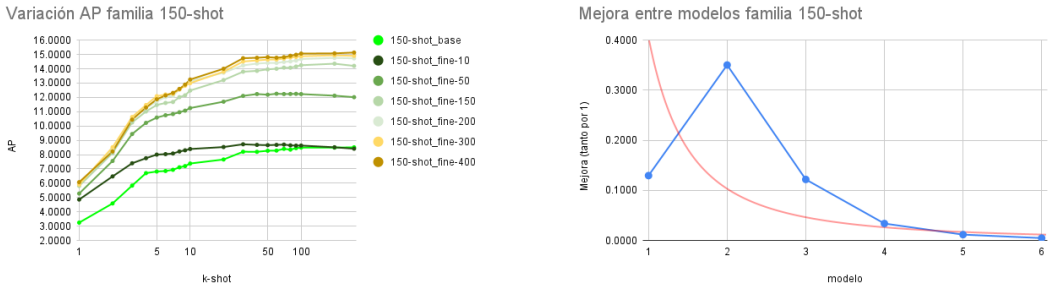
(a) (b)

Figura 4.4: a) AP para la familia de modelos 10-shot. b) Ganancia entre modelos consecutivos (la curva roja es la línea de tendencia).



(a) (b)

Figura 4.5: a) AP para la familia de modelos 50-shot. b) Ganancia entre modelos consecutivos (la curva roja es la línea de tendencia).



(a) (b)

Figura 4.6: a) AP para la familia de modelos 150-shot. b) Ganancia entre modelos consecutivos (la curva roja es la línea de tendencia).

La figura 4.7 recoge una comparativa de todos los modelos entrenados. En ella se observa que los modelos base son los que presentan unos peores resultados. De hecho, *10-shot_fine-10* alcanza el mismo desempeño que *150-shot_base* con un coste computacional mucho menor.

Para valores de *finetuning* inferiores a 150 se observa la existencia de dos grupos, asociados en torno al valor de *F-shot*. Es decir, contando desde *10-shot_fine-10* (segundo línea en tono azul), se ve que los modelos sobre los que se realizó *finetuning* de 10, son peores que los modelos sobre los que se realizó *finetuning* de 50. Dentro del primer grupo, el modelo entrenado de base con *10-shot* es peor que el de *50-shot* y, a su vez, este es peor que el de *150-shot*. Para el segundo grupo, la posición de *150-shot* y *10-shot* se alternan.

Para valores de *finetuning* superiores a 150, se observa que los modelos derivados de *50-shot* (tonos lila) son los que presentan un mejor desempeño, seguidos de la familia *150-shot* (tonos ocres).

De hecho, el modelo que alcanza un mayor valor de AP es *50-shot_fine-400* para un *k-shot* de 200. Esto es un indicador de que, aunque *150-shot_base* funciona mejor que *50-shot_base*, *50-shot_base* alcanza un mejor rendimiento cuando se reentrena con la información de soporte.

Los modelos de la familia *10-shot* para *F-shot* grandes nunca llegan a alcanzar un rendimiento significativo. Particularmente, *10-shot_fine-300* es incluso inferior a *150-shot_fine-150*.

Si nos fijamos en la zona del espacio ausente de resultados, es fácil ver que el incremento más pronunciado en el desempeño ocurre cuando se realiza *finetuning* con *F-shot* de 50 y superiores. Haciendo una comparación con las figuras 4.4 b), 4.5 b) y 4.6 b) esto coincide con el pico de beneficio (punto 2 en el eje de abscisas).

Por último, al comprar *150-shot_base* con *150-shot_fine-10* y *10-shot_base* con *10-shot_fine-10*, se advierte que la diferencia entre los primeros es mucho menos acentuada que la diferencia entre los segundos (el beneficio del primer *finetuning* sobre el modelo 150-shot apenas existe).

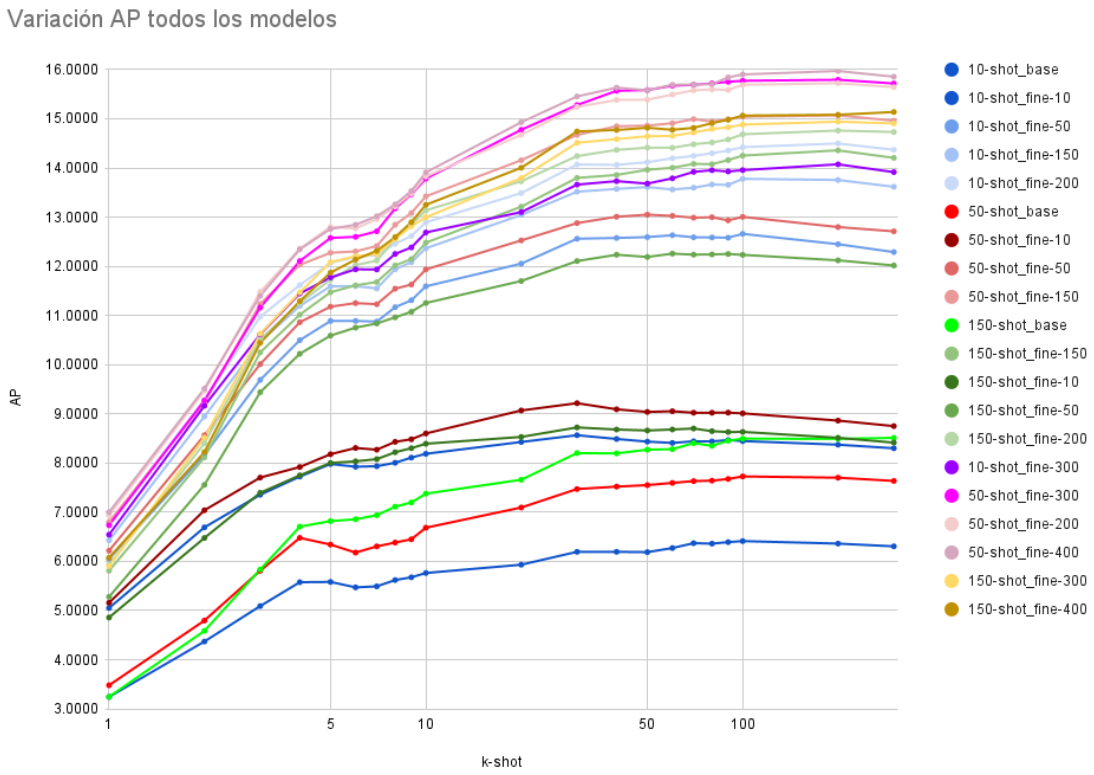


Figura 4.7: Comparativa AP de todos los modelos entrenados. Familia 10-shot tonos azules y violeta. Familia 50-shot tonos rojos y lilas. Familia 150-shot tonos verdes y ocres.

4.4. Experimento 3: Variación entre clases

El objetivo es determinar cuanto varía el rendimiento al evaluar sobre categorías más o menos parecidas a las de entrenamiento.

Para ello se toma el modelo *10-shot_fine-10*, evaluado sobre un *k-shot* de 10, y se comparan cualitativamente los resultados entre clases. Es decir, se observa si las clases de test (C_{nuevas}) con mayor AP, son también las más parecidas a las clases de entrenamiento (C_{base}).

El motivo de elegir únicamente un modelo para realizar este experimento, deriva de la necesidad de analizar los datos entre clases. Es decir, no es interesante conocer como varía el AP de clase entre modelos, sino saber qué clases reportan un mayor AP (son las mismas para todos los modelos) y su similitud con las categorías de entrenamiento.

La tabla 4.6 recoge las categorías que integran las particiones de entrenamiento y test.

Entrenamiento (C_{base})			**Test** (C_{nuevas})		
truck	*baseball bat*	*cake*	*person*	*boat*	*bottle*
traffic light	*baseball glove*	*bed*	*bicycle*	*bird*	*chair*
fire hydrant	*skateboard*	*toilet*	*car*	*cat*	*couch*
stop sign	*surfboard*	*laptop*	*motorcycle*	*dog*	*potted plant*
parking meter	*tennis racket*	*mouse*	*airplane*	*horse*	*dining table*
bench	*wine glass*	*remote*	*bus*	*sheep*	*tv*
elephant	*cup*	*keyboard*	*train*	*cow*	
bear	*fork*	*cell phone*			
zebra	*knife*	*microwave*			
giraffe	*spoon*	*oven*			
backpack	*bowl*	*toaster*			
umbrella	*banana*	*sink*			
handbag	*apple*	*refrigerator*			
tie	*sandwich*	*book*			
suitcase	*orange*	*clock*			
frisbee	*broccoli*	*vase*			
skis	*carrot*	*scissors*			
snowboard	*hot dog*	*teddy bear*			
sports ball	*pizza*	*hair drier*			
kite	*donut*	*toothbrush*			

Tabla 4.6: Distribución de clases entrenamiento/test

La tabla 4.7 muestra el AP obtenido por el modelo en cada una de las clases que integran el conjunto de test. Se puede observar que existen 6 clases con valores de AP atípicamente elevados: *airplane*, *bus*, *train*, *cat*, *horse* y *tv*. Haciendo una comparación cualitativa con la tabla 4.6 es fácil ver que algunas de ellas son muy parecidas a categorías de entrenamiento:

- La clase *tv* con AP 21.58, es parecida a la clase de entrenamiento *laptop*.
- Las clases *bus* y *train* con AP 20.27 y 15.56 respectivamente, son parecidas a la clase de entrenamiento *truck*.
- La clase *horse* con AP 11.7 es parecida a la clase de entrenamiento *zebra*.

Clase	AP	Clase	AP	Clase	AP
person	2.63	*boat*	1.07	*bottle*	4.03
bicycle	3.98	*bird*	1.89	*chair*	0.55
car	4.47	*cat*	21.34	*couch*	7.72
motorcycle	5.27	*dog*	9.94	*potted plant*	1.29
airplane	16.06	*horse*	11.70	*dining table*	5.38
bus	20.27	*sheep*	3.05	*tv*	21.58
train	15.56	*cow*	5.98	**AP Promedio**	**8.19**

Tabla 4.7: Resultado AP del modelo *10-shot_fine-10* con *k-shot* 10 por clase.

4.5. Experimento 4: FPN

El objetivo de este experimento es analizar la mejora que supone la introducción de una FPN. Para ello se toma el modelo *10-shot_fine-10_FPN* y se compara con su análogo sin FPN para los siguientes tamaños de *k-shot*: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, y 300.

La figura 4.8 compara el rendimiento de la versión con FPN respecto a la versión sin FPN para distintos tamaños de *k-shot*. Puede observarse que el uso de FPN mejora el rendimiento de la red para todos los tamaños de soporte. Aunque para 10 *k-shot* la diferencia entre ambas versiones es solo de 0.3 puntos; en general, para *k-shot* inferiores a 5 la diferencia oscila entre 1 y 1.5 puntos y para *k-shot* superiores, entre 0.5 y 0.8.

La figura 4.9 recoge esta misma comparativa, desagregada para objetos pequeños (APs), medianos (APm) y grandes (APl). En ella se ve como la introducción de la FPN supone un mayor beneficio en el grupo de objetos pequeños, que no son detectados por FewX original (APs sin FPN entre 0.8 y 1.1). Concretamente, el beneficio entre ambas versiones es considerablemente elevado, fluctuando entre 2 y 3 puntos.

El segundo grupo donde mayor beneficio supone la introducción de la FPN es en el de objetos medianos, presentando una mejora entre 1.5 y 2 puntos.

Finalmente, en el grupo de objetos grandes, para *k-shot* inferiores a 10 sí que se presenta cierto grado de beneficio (en torno 1 punto). Sin embargo, para *k-shot* superiores ambos modelos muestran un desempeño similar.

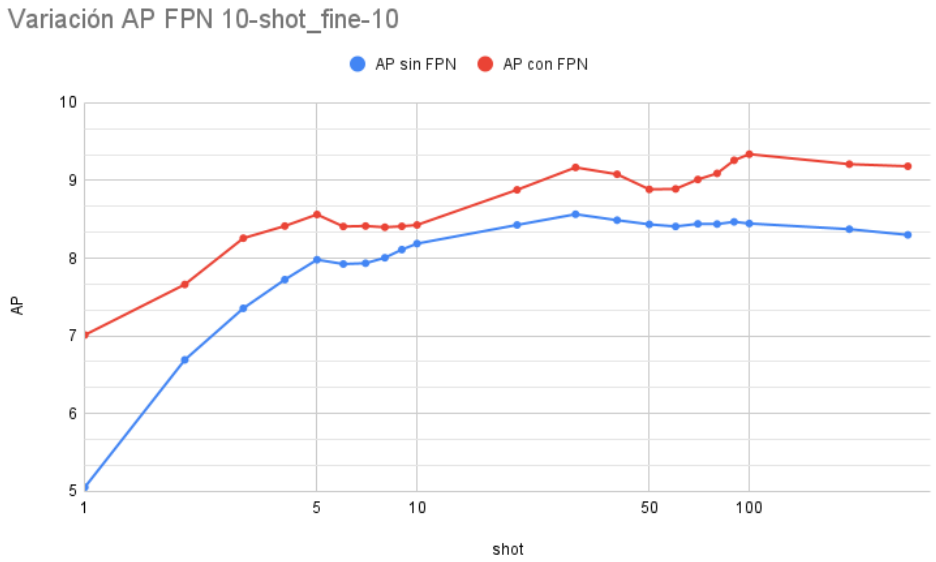


Figura 4.8: Comparativa AP entre la versión con FPN frente a la versión sin FPN para distintos tamaños de *k-shot*.

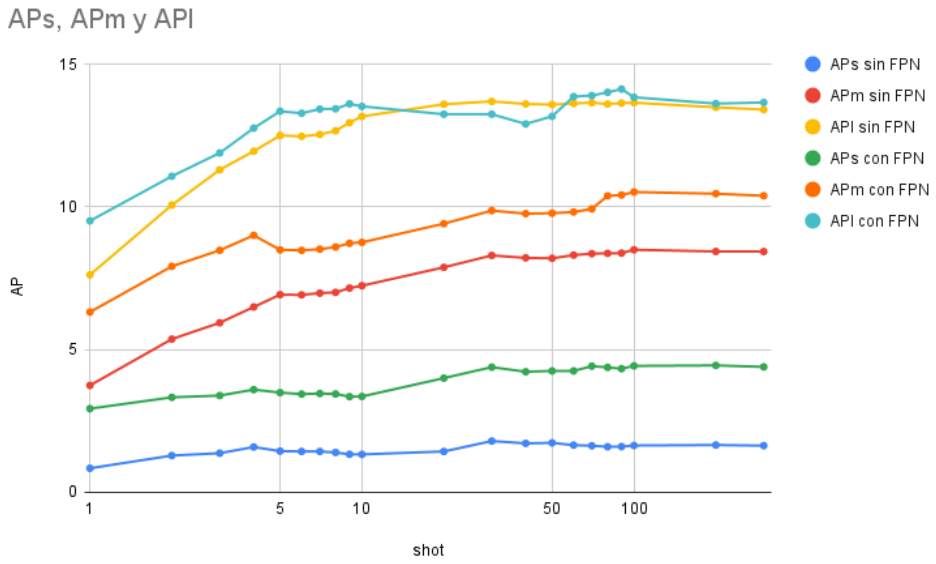


Figura 4.9: Comparativa APs, APm y APl entre la versión con FPN frente a la versión sin FPN para distintos tamaños de *k-shot*

Capítulo 5

Discusión de resultados

Analizando los resultados obtenidos en el desarrollo experimental se puede concluir que:

1. **Existe variabilidad entre soportes**. Al analizar el desempeño del modelo *10-shot_base* sobre 40 soportes disjuntos obtuvimos una desviación de 0.44 puntos de AP. Una desviación de esta magnitud en un entorno donde los resultados oscilan alrededor de 10 puntos es una desviación considerablemente alta.

2. **La variabilidad entre soportes depende exclusivamente de las imágenes empleadas** y es uniforme entre configuraciones de modelos. Es decir, ni el número de imágenes de soporte utilizadas para entrenar el modelo base, ni el hecho de haber realizado (o no) *finetuning* influye en la variabilidad. Esta solamente depende del número y características de las imágenes de soporte empleadas en test.

3. **El AP crece con *k-shot* hasta llegar a un umbral de saturación**. Este umbral depende directamente del número de imágenes de soporte que se usaron durante el entrenamiento base del modelo y de *F-shot*. Cuanto mayores son estos elementos, mayor es el AP alcanzado y más tarde se alcanza la saturación.

4. **El AP crece con *F-shot***. Todos los modelos base mejoran su rendimiento cuando se aplica sobre ellos la técnica del *finetuning*. Sin embargo, el beneficio obtenido es menor a medida que aumenta *F-shot*.

5. Si se dispone de imágenes etiquetadas suficientes, **hacer *finetuning* es siempre mejor opción que entrenar un modelo base con un *shot* grande**. El motivo es que se alcanza un mejor desempeño con un coste computacional muy inferior.

6. El objetivo de esta red es detectar objetos de categorías nunca vistas dado un conjunto reducido de ejemplos etiquetados. Sin embargo, **cuanto mayor sea el parecido de estas categorías respecto a las de entrenamiento, mayor será el desempeño de la red**.

7. La introducción de una **FPN mejora la calidad de FewX**. Particularmente, facilita la detección de objetos pequeños y medianos obteniendo un mayor AP en estos casos. Para objetos grandes, el rendimiento es análogo a la versión de FewX sin FPN.

Relacionando estas conclusiones con las hipótesis de partida (apartado 1.2) se puede afirmar que **todas ellas quedan contrastadas**. La hipótesis 1, mayor rendimiento cuando las clases de entrenamiento y de test son parecidas, con la conclusión 6. La hipótesis 2, mejor rendimiento cuanto mayor es el soporte, con la conclusión 3. La hipótesis 3, mayor rendimiento al realizar *finetuning*, con la conclusión 4. La hipótesis 4, uso de FPN hace las red más robustas ante la detección de objetos de distinto tamaños, con la conclusión 7. El resto de inferencias matizan y amplían los resultados del estudio.

La tabla 5.1 muestra una comparativa de los resultados obtenidos respecto a otras aproximaciones del estado del conocimiento.

Método	AP	AP50	AP75	APs	APm	APl
LSTD [29]	3.2	8.1	2.1	0.9	2.0	6.5
TFA [34]	9.1	17.3	8.5	-	-	-
MPSR [33]	**9.8**	**17.9**	**9.7**	**3.3**	**9.2**	**16.1**
META-YOLO [28]	5.6	12.3	4.6	0.9	3.5	10.5
Meta Det [35]	7.1	14.6	6.1	1.0	4.1	12.2
Meta R-CNN [30]	8.7	19.1	6.6	2.3	7.7	14.0
FewX [1]	**8.1**	**16.2**	**7.2**	**1.3**	**7.2**	**13.1**
Nuestro	**9.2**	**18.0**	**7.5**	**3.1**	**8.7**	**13.5**

Tabla 5.1: Comparativa entre métodos similares del estado del conocimiento. Los resultados fueron obtenidos sobre las 20 clases nuevas de MS COCO *val2017* utilizando un *k-shot* de 10.

En ella se muestra como las modificaciones propuestas sobre *FewX* mejoran significativamente la precisión con respecto a la red base. Esta diferencia es especialmente importante para el caso de objetos pequeños (APs, objetos con tamaño de hasta 1024 píxeles) y medianos (APm, entre 1024 y 9216 píxeles). Además, los resultados obtenidos en estas métricas están en línea con las mejores aproximaciones del estado del conocimiento. Esto incluye trabajos recientes como MPSR

que, a pesar de mejorar ligeramente los resultados de FewX (método en el que se basa este trabajo), obtiene APs comparables a los de nuestra propuesta gracias a las mejoras introducidas en este trabajo.

La detección de objetos pequeños es una de la tareas más complejas dentro de la detección de objetos y continúa siendo un campo de investigación muy activo. Los resultados obtenidos prueban que las mejoras propuestas son válidas para aumentar la precisión en el aprendizaje basado en *few-shot*.

Capítulo 6

Conclusiones y posibles ampliaciones

En este trabajo se ha estudiado el problema de la detección de objetos *few-shot* en entornos de escasez de datos. Tomando como punto de partida la red FewX, se ha analizado experimentalmente la influencia de los parámetros: *k-shot*, *F-shot*, similitud entre clases de entrenamiento/test y soportes. Es decir, se ha cuantificado hasta que punto la modificación de estos elementos afecta al desempeño de la red.

Así, se ha observado que el rendimiento de la red aumenta con *k-shot* y *F-shot* hasta llegar a un umbral a partir del cual el agregado de información de soporte deja de concentrar información relevante; que testear sobre clases parecidas a las de entrenamiento implica obtener unos mejores resultados; y que la variabilidad entre conjuntos de soporte es alta.

Una vez adquirida una profunda comprensión acerca de este tipo de arquitecturas, en este TFG se han implementado una serie de mejoras sobre la red FewX. Concretamente, modificando el manejo del conjunto de soporte se ha conseguido un sistema escalable ante cualquier número de clases y tamaño de *k-shot*. Por otro lado, los tiempos de inferencia se optimizaron reduciendo la transferencia de datos entre la memoria de CPU y GPU.

Por último, la introducción de una *Feature Pyramid Network* ha permitido aumentar la robustez de la red ante objetos de diferentes tamaños. La aplicación de estas mejoras supone una diferencia significativa tanto en objetos pequeños como medianos obteniendo resultados similares a la versión base en objetos grandes.

Aunque los resultados obtenidos por los detectores *few-shot* son prometedores, todavía están lejos de alcanzar precisiones similares a las de los métodos tradicionales. Su principal ventaja es la capacidad de detectar objetos de categorías nunca vistas a partir de muy pocos ejemplos etiquetados. Esto es útil en entornos

de escasez de datos, donde la información etiquetada puede ser muy costosa de obtener. Sin embargo, en escenarios en los que se dispone de suficientes datos etiquetados, la mejor opción es entrenar un detector tradicional basado en CNN.

Una ampliación interesante para este trabajo sería repetir el estudio experimental sobre otras aproximaciones *few-shot* como Meta R-CNN [30], Meta-DETR [4], Meta Det [35], META-YOLO [28], etc. y observar si el comportamiento observado es generalizable.

También sería interesante profundizar en redes *few-shot* no basadas en regiones de interés, pues Meta-DETR [4] pone de manifiesto que las aproximaciones que unifican el meta-aprendizaje de la localización y clasificación del objeto al nivel de la imagen presentan unos buenos resultados.

Apéndice A

Distribución imágenes MS COCO

Distribución de objetos en la partición de **entrenamiento** (80 clases, 860001 objetos):

id	**clase**	**nº objetos**	**id**	**clase**	**nº objetos**
1	person	262465	46	wine glass	7913
2	bicycle	7113	47	cup	20650
3	car	43867	48	fork	5479
4	motorcycle	8725	49	knife	7770
5	airplane	5135	50	spoon	6165
6	bus	6069	51	bowl	14358
7	train	4571	52	banana	9458
8	truck	9973	53	apple	5851
9	boat	10759	54	sandwich	4373
10	traffic light	12884	55	orange	6399
11	fire hydrant	1865	56	broccoli	7308
13	stop sign	1983	57	carrot	7852
14	parking meter	1285	58	hot dog	2918
15	bench	9838	59	pizza	5821
16	bird	10806	60	donut	7179
17	cat	4768	61	cake	6353
18	dog	5508	62	chair	38491
19	horse	6587	63	couch	5779
20	sheep	9509	64	potted plant	8652
21	cow	8147	65	bed	4192
22	elephant	5513	67	dining table	15714
23	bear	1294	70	toilet	4157

24	zebra	5303	72	tv	5805
25	giraffe	5131	73	laptop	4970
27	backpack	8720	74	mouse	2262
28	umbrella	11431	75	remote	5703
31	handbag	12354	76	keyboard	2855
32	tie	6496	77	cell phone	6434
33	suitcase	6192	78	microwave	1673
34	frisbee	2682	79	oven	3334
35	skis	6646	80	toaster	225
36	snowboard	2685	81	sink	5610
37	sports ball	6347	82	refrigerator	2637
38	kite	9076	84	book	24715
39	baseball bat	3276	85	clock	6334
40	baseball glove	3747	86	vase	6613
41	skateboard	5543	87	scissors	1481
42	surfboard	6126	88	teddy bear	4793
43	tennis racket	4812	89	hair drier	198
44	bottle	24342	90	toothbrush	1954

Distribución de objetos en la partición de **validación** (80 clases, 36 781 objetos):

id	**clase**	**nº objetos**	**id**	**clase**	**nº objetos**
1	person	11004	46	wine glass	343
2	bicycle	316	47	cup	899
3	car	1932	48	fork	215
4	motorcycle	371	49	knife	326
5	airplane	143	50	spoon	253
6	bus	285	51	bowl	626
7	train	190	52	banana	379
8	truck	415	53	apple	239
9	boat	430	54	sandwich	177
10	traffic light	637	55	orange	287
11	fire hydrant	101	56	broccoli	316
13	stop sign	75	57	carrot	371
14	parking meter	60	58	hot dog	127
15	bench	413	59	pizza	285

16	bird	440	60	donut	338
17	cat	202	61	cake	316
18	dog	218	62	chair	1791
19	horse	273	63	couch	261
20	sheep	361	64	potted plant	343
21	cow	380	65	bed	163
22	elephant	255	67	dining table	697
23	bear	71	70	toilet	179
24	zebra	268	72	tv	288
25	giraffe	232	73	laptop	231
27	backpack	371	74	mouse	106
28	umbrella	413	75	remote	283
31	handbag	540	76	keyboard	153
32	tie	254	77	cell phone	262
33	suitcase	303	78	microwave	55
34	frisbee	115	79	oven	143
35	skis	241	80	toaster	9
36	snowboard	69	81	sink	225
37	sports ball	263	82	refrigerator	126
38	kite	336	84	book	1161
39	baseball bat	146	85	clock	267
40	baseball glove	148	86	vase	277
41	skateboard	179	87	scissors	36
42	surfboard	269	88	teddy bear	191
43	tennis racket	225	89	hair drier	11
44	bottle	1025	90	toothbrush	57

Apéndice B

Arquitectura FewX con FPN

```
FsodRCNN(
  (backbone): FPN(
    (fpn_lateral2): Conv2d(256, 256, kernel_size=(1, 1),
       stride=(1, 1))
    (fpn_output2): Conv2d(256, 256, kernel_size=(3, 3),
       stride=(1, 1), padding=(1, 1))
    (fpn_lateral3): Conv2d(512, 256, kernel_size=(1, 1),
       stride=(1, 1))
    (fpn_output3): Conv2d(256, 256, kernel_size=(3, 3),
       stride=(1, 1), padding=(1, 1))
    (fpn_lateral4): Conv2d(1024, 256, kernel_size=(1, 1),
       stride=(1, 1))
    (fpn_output4): Conv2d(256, 256, kernel_size=(3, 3),
       stride=(1, 1), padding=(1, 1))
    (fpn_lateral5): Conv2d(2048, 256, kernel_size=(1, 1),
       stride=(1, 1))
    (fpn_output5): Conv2d(256, 256, kernel_size=(3, 3),
       stride=(1, 1), padding=(1, 1))
    (top_block): LastLevelMaxPool()
    (bottom_up): ResNet(
      (stem): BasicStem(
        (conv1): Conv2d(
          3, 64, kernel_size=(7, 7), stride=(2, 2),
             padding=(3, 3), bias=False
          (norm): FrozenBatchNorm2d(num_features=64, eps=1
             e-05)
        )
      )
      (res2): Sequential(
```

```
        (0): BottleneckBlock(
          (shortcut): Conv2d(
            64, 256, kernel_size=(1, 1), stride=(1, 1),
               bias=False
            (norm): FrozenBatchNorm2d(num_features=256,
               eps=1e-05)
          )
          (conv1): Conv2d(
            64, 64, kernel_size=(1, 1), stride=(1, 1),
               bias=False
            (norm): FrozenBatchNorm2d(num_features=64, eps
               =1e-05)
          )
          (conv2): Conv2d(
            64, 64, kernel_size=(3, 3), stride=(1, 1),
               padding=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=64, eps
               =1e-05)
          )
          (conv3): Conv2d(
            64, 256, kernel_size=(1, 1), stride=(1, 1),
               bias=False
            (norm): FrozenBatchNorm2d(num_features=256,
               eps=1e-05)
          )
        )
        (1): BottleneckBlock(
          (conv1): Conv2d(
            256, 64, kernel_size=(1, 1), stride=(1, 1),
               bias=False
            (norm): FrozenBatchNorm2d(num_features=64, eps
               =1e-05)
          )
          (conv2): Conv2d(
            64, 64, kernel_size=(3, 3), stride=(1, 1),
               padding=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=64, eps
               =1e-05)
          )
          (conv3): Conv2d(
            64, 256, kernel_size=(1, 1), stride=(1, 1),
               bias=False
            (norm): FrozenBatchNorm2d(num_features=256,
```

```
            eps=1e-05)
        )
      )
      (2): BottleneckBlock(
        (conv1): Conv2d(
          256, 64, kernel_size=(1, 1), stride=(1, 1),
            bias=False
          (norm): FrozenBatchNorm2d(num_features=64, eps
            =1e-05)
        )
        (conv2): Conv2d(
          64, 64, kernel_size=(3, 3), stride=(1, 1),
            padding=(1, 1), bias=False
          (norm): FrozenBatchNorm2d(num_features=64, eps
            =1e-05)
        )
        (conv3): Conv2d(
          64, 256, kernel_size=(1, 1), stride=(1, 1),
            bias=False
          (norm): FrozenBatchNorm2d(num_features=256,
            eps=1e-05)
        )
      )
    )
    (res3): Sequential(
      (0): BottleneckBlock(
        (shortcut): Conv2d(
          256, 512, kernel_size=(1, 1), stride=(2, 2),
            bias=False
          (norm): FrozenBatchNorm2d(num_features=512,
            eps=1e-05)
        )
        (conv1): Conv2d(
          256, 128, kernel_size=(1, 1), stride=(2, 2),
            bias=False
          (norm): FrozenBatchNorm2d(num_features=128,
            eps=1e-05)
        )
        (conv2): Conv2d(
          128, 128, kernel_size=(3, 3), stride=(1, 1),
            padding=(1, 1), bias=False
          (norm): FrozenBatchNorm2d(num_features=128,
            eps=1e-05)
```

```
        )
        (conv3): Conv2d(
          128, 512, kernel_size=(1, 1), stride=(1, 1),
             bias=False
          (norm): FrozenBatchNorm2d(num_features=512,
             eps=1e-05)
        )
      )
      (1): BottleneckBlock(
        (conv1): Conv2d(
          512, 128, kernel_size=(1, 1), stride=(1, 1),
             bias=False
          (norm): FrozenBatchNorm2d(num_features=128,
             eps=1e-05)
        )
        (conv2): Conv2d(
          128, 128, kernel_size=(3, 3), stride=(1, 1),
             padding=(1, 1), bias=False
          (norm): FrozenBatchNorm2d(num_features=128,
             eps=1e-05)
        )
        (conv3): Conv2d(
          128, 512, kernel_size=(1, 1), stride=(1, 1),
             bias=False
          (norm): FrozenBatchNorm2d(num_features=512,
             eps=1e-05)
        )
      )
      (2): BottleneckBlock(
        (conv1): Conv2d(
          512, 128, kernel_size=(1, 1), stride=(1, 1),
             bias=False
          (norm): FrozenBatchNorm2d(num_features=128,
             eps=1e-05)
        )
        (conv2): Conv2d(
          128, 128, kernel_size=(3, 3), stride=(1, 1),
             padding=(1, 1), bias=False
          (norm): FrozenBatchNorm2d(num_features=128,
             eps=1e-05)
        )
        (conv3): Conv2d(
          128, 512, kernel_size=(1, 1), stride=(1, 1),
```

```
            bias=False
          (norm): FrozenBatchNorm2d(num_features=512,
            eps=1e-05)
        )
      )
      (3): BottleneckBlock(
        (conv1): Conv2d(
          512, 128, kernel_size=(1, 1), stride=(1, 1),
            bias=False
          (norm): FrozenBatchNorm2d(num_features=128,
            eps=1e-05)
        )
        (conv2): Conv2d(
          128, 128, kernel_size=(3, 3), stride=(1, 1),
            padding=(1, 1), bias=False
          (norm): FrozenBatchNorm2d(num_features=128,
            eps=1e-05)
        )
        (conv3): Conv2d(
          128, 512, kernel_size=(1, 1), stride=(1, 1),
            bias=False
          (norm): FrozenBatchNorm2d(num_features=512,
            eps=1e-05)
        )
      )
    )
    (res4): Sequential(
      (0): BottleneckBlock(
        (shortcut): Conv2d(
          512, 1024, kernel_size=(1, 1), stride=(2, 2),
            bias=False
          (norm): FrozenBatchNorm2d(num_features=1024,
            eps=1e-05)
        )
        (conv1): Conv2d(
          512, 256, kernel_size=(1, 1), stride=(2, 2),
            bias=False
          (norm): FrozenBatchNorm2d(num_features=256,
            eps=1e-05)
        )
        (conv2): Conv2d(
          256, 256, kernel_size=(3, 3), stride=(1, 1),
            padding=(1, 1), bias=False
```

```
            (norm): FrozenBatchNorm2d(num_features=256,
                eps=1e-05)
          )
          (conv3): Conv2d(
            256, 1024, kernel_size=(1, 1), stride=(1, 1),
                bias=False
            (norm): FrozenBatchNorm2d(num_features=1024,
                eps=1e-05)
          )
        )
        (1): BottleneckBlock(
          (conv1): Conv2d(
            1024, 256, kernel_size=(1, 1), stride=(1, 1),
                bias=False
            (norm): FrozenBatchNorm2d(num_features=256,
                eps=1e-05)
          )
          (conv2): Conv2d(
            256, 256, kernel_size=(3, 3), stride=(1, 1),
                padding=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=256,
                eps=1e-05)
          )
          (conv3): Conv2d(
            256, 1024, kernel_size=(1, 1), stride=(1, 1),
                bias=False
            (norm): FrozenBatchNorm2d(num_features=1024,
                eps=1e-05)
          )
        )
        (2): BottleneckBlock(
          (conv1): Conv2d(
            1024, 256, kernel_size=(1, 1), stride=(1, 1),
                bias=False
            (norm): FrozenBatchNorm2d(num_features=256,
                eps=1e-05)
          )
          (conv2): Conv2d(
            256, 256, kernel_size=(3, 3), stride=(1, 1),
                padding=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=256,
                eps=1e-05)
          )
```

```
      (conv3): Conv2d(
        256, 1024, kernel_size=(1, 1), stride=(1, 1),
           bias=False
        (norm): FrozenBatchNorm2d(num_features=1024,
           eps=1e-05)
      )
    )
    (3): BottleneckBlock(
      (conv1): Conv2d(
        1024, 256, kernel_size=(1, 1), stride=(1, 1),
           bias=False
        (norm): FrozenBatchNorm2d(num_features=256,
           eps=1e-05)
      )
      (conv2): Conv2d(
        256, 256, kernel_size=(3, 3), stride=(1, 1),
           padding=(1, 1), bias=False
        (norm): FrozenBatchNorm2d(num_features=256,
           eps=1e-05)
      )
      (conv3): Conv2d(
        256, 1024, kernel_size=(1, 1), stride=(1, 1),
           bias=False
        (norm): FrozenBatchNorm2d(num_features=1024,
           eps=1e-05)
      )
    )
    (4): BottleneckBlock(
      (conv1): Conv2d(
        1024, 256, kernel_size=(1, 1), stride=(1, 1),
           bias=False
        (norm): FrozenBatchNorm2d(num_features=256,
           eps=1e-05)
      )
      (conv2): Conv2d(
        256, 256, kernel_size=(3, 3), stride=(1, 1),
           padding=(1, 1), bias=False
        (norm): FrozenBatchNorm2d(num_features=256,
           eps=1e-05)
      )
      (conv3): Conv2d(
        256, 1024, kernel_size=(1, 1), stride=(1, 1),
           bias=False
```

```
            (norm): FrozenBatchNorm2d(num_features=1024,
                eps=1e-05)
          )
        )
        (5): BottleneckBlock(
          (conv1): Conv2d(
            1024, 256, kernel_size=(1, 1), stride=(1, 1),
                bias=False
            (norm): FrozenBatchNorm2d(num_features=256,
                eps=1e-05)
          )
          (conv2): Conv2d(
            256, 256, kernel_size=(3, 3), stride=(1, 1),
                padding=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=256,
                eps=1e-05)
          )
          (conv3): Conv2d(
            256, 1024, kernel_size=(1, 1), stride=(1, 1),
                bias=False
            (norm): FrozenBatchNorm2d(num_features=1024,
                eps=1e-05)
          )
        )
      )
      (res5): Sequential(
        (0): BottleneckBlock(
          (shortcut): Conv2d(
            1024, 2048, kernel_size=(1, 1), stride=(2, 2),
                bias=False
            (norm): FrozenBatchNorm2d(num_features=2048,
                eps=1e-05)
          )
          (conv1): Conv2d(
            1024, 512, kernel_size=(1, 1), stride=(2, 2),
                bias=False
            (norm): FrozenBatchNorm2d(num_features=512,
                eps=1e-05)
          )
          (conv2): Conv2d(
            512, 512, kernel_size=(3, 3), stride=(1, 1),
                padding=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=512,
```

```
            eps=1e-05)
        )
        (conv3): Conv2d(
          512, 2048, kernel_size=(1, 1), stride=(1, 1),
            bias=False
          (norm): FrozenBatchNorm2d(num_features=2048,
            eps=1e-05)
        )
      )
      (1): BottleneckBlock(
        (conv1): Conv2d(
          2048, 512, kernel_size=(1, 1), stride=(1, 1),
            bias=False
          (norm): FrozenBatchNorm2d(num_features=512,
            eps=1e-05)
        )
        (conv2): Conv2d(
          512, 512, kernel_size=(3, 3), stride=(1, 1),
            padding=(1, 1), bias=False
          (norm): FrozenBatchNorm2d(num_features=512,
            eps=1e-05)
        )
        (conv3): Conv2d(
          512, 2048, kernel_size=(1, 1), stride=(1, 1),
            bias=False
          (norm): FrozenBatchNorm2d(num_features=2048,
            eps=1e-05)
        )
      )
      (2): BottleneckBlock(
        (conv1): Conv2d(
          2048, 512, kernel_size=(1, 1), stride=(1, 1),
            bias=False
          (norm): FrozenBatchNorm2d(num_features=512,
            eps=1e-05)
        )
        (conv2): Conv2d(
          512, 512, kernel_size=(3, 3), stride=(1, 1),
            padding=(1, 1), bias=False
          (norm): FrozenBatchNorm2d(num_features=512,
            eps=1e-05)
        )
        (conv3): Conv2d(
```

```
            512, 2048, kernel_size=(1, 1), stride=(1, 1),
               bias=False
            (norm): FrozenBatchNorm2d(num_features=2048,
               eps=1e-05)
          )
        )
      )
    )
  )
  (proposal_generator): FsodRPN(
    (rpn_head): StandardRPNHead(
      (conv): Conv2d(256, 256, kernel_size=(3, 3), stride
         =(1, 1), padding=(1, 1))
      (objectness_logits): Conv2d(256, 3, kernel_size=(1,
         1), stride=(1, 1))
      (anchor_deltas): Conv2d(256, 12, kernel_size=(1, 1),
          stride=(1, 1))
    )
    (anchor_generator): DefaultAnchorGenerator(
      (cell_anchors): BufferList()
    )
  )
  (roi_heads): FsodStandardROIHeads(
    (box_pooler): ROIPooler(
      (level_poolers): ModuleList(
        (0): ROIAlign(output_size=(7, 7), spatial_scale
           =0.25, sampling_ratio=0, aligned=True)
        (1): ROIAlign(output_size=(7, 7), spatial_scale
           =0.125, sampling_ratio=0, aligned=True)
        (2): ROIAlign(output_size=(7, 7), spatial_scale
           =0.0625, sampling_ratio=0, aligned=True)
        (3): ROIAlign(output_size=(7, 7), spatial_scale
           =0.03125, sampling_ratio=0, aligned=True)
      )
    )
    (box_head): PersonalBoxHead(
      (conv1): Conv2d(
        256, 2048, kernel_size=(3, 3), stride=(1, 1),
           padding=(1, 1), bias=False
        (norm): BatchNorm2d(2048, eps=1e-05, momentum=0.1,
            affine=True, track_running_stats=True)
      )
      (conv2): Conv2d(
```

```
        2048, 2048, kernel_size=(3, 3), stride=(1, 1),
            padding=(1, 1), bias=False
        (norm): BatchNorm2d(2048, eps=1e-05, momentum=0.1,
            affine=True, track_running_stats=True)
      )
    )
    (box_predictor): FsodFastRCNNOutputLayers(
      (conv_1): Conv2d(4096, 512, kernel_size=(1, 1),
          stride=(1, 1), bias=False)
      (conv_2): Conv2d(512, 512, kernel_size=(3, 3),
          stride=(1, 1), bias=False)
      (conv_3): Conv2d(512, 2048, kernel_size=(1, 1),
          stride=(1, 1), bias=False)
      (bbox_pred_pr): Linear(in_features=2048,
          out_features=4, bias=True)
      (cls_score_pr): Linear(in_features=2048,
          out_features=2, bias=True)
      (conv_cor): Conv2d(2048, 2048, kernel_size=(1, 1),
          stride=(1, 1), bias=False)
      (cls_score_cor): Linear(in_features=2048,
          out_features=2, bias=True)
      (fc_1): Linear(in_features=4096, out_features=2048,
          bias=True)
      (fc_2): Linear(in_features=2048, out_features=2048,
          bias=True)
      (cls_score_fc): Linear(in_features=2048,
          out_features=2, bias=True)
      (avgpool): AvgPool2d(kernel_size=3, stride=1,
          padding=0)
      (avgpool_fc): AvgPool2d(kernel_size=7, stride=7,
          padding=0)
    )
  )
)
```

Apéndice C

Manual de usuario

Para la ejecución de los experimentos es necesario **disponer de un contenedor docker con detectron2 0.3** instalado. El dockerfile que lo genera es:

```
FROM nvidia/cuda:10.1-cudnn7-devel

ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && apt-get install -y \
    python3-opencv ca-certificates python3-dev git wget sudo ninja-build
RUN ln -sv /usr/bin/python3 /usr/bin/python

# crear un usuario
ARG USER_ID=1000
RUN useradd -m --no-log-init --system  --uid ${USER_ID} appuser -g sudo
RUN echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers
USER appuser
WORKDIR /home/appuser

ENV PATH="/home/appuser/.local/bin:${PATH}"
RUN wget https://bootstrap.pypa.io/get-pip.py && \
    python3 get-pip.py --user && \
    rm get-pip.py

# instalar las dependencias
RUN pip install --user tensorboard cmake
RUN pip install --user torch==1.7.1 torchvision==0.8.2 -f \
    https://download.pytorch.org/whl/cu101/torch_stable.html
RUN pip install pandas
RUN pip install scikit-image

RUN pip install --user 'git+https://github.com/facebookresearch/fvcore'

#instalamos detectron2 para CUDA
RUN git clone https://github.com/facebookresearch/detectron2 detectron2_repo
ENV FORCE_CUDA="1"
ARG TORCH_CUDA_ARCH_LIST="Kepler;Kepler+Tesla;Maxwell;Maxwell+Tegra;Pascal;Volta;Turing"
ENV TORCH_CUDA_ARCH_LIST="${TORCH_CUDA_ARCH_LIST}"

RUN pip install --user -e detectron2_repo

# Configuración adicional
ENV FVCORE_CACHE="/tmp"
WORKDIR /home/appuser/detectron2_repo
```

Si la creación es correcta, al ejecutar *python -m detectron2.utils.collect_env* debería recolectarse la información del entorno, el cual debe ser:

```
[32m[04/14 22:00:29 detectron2]: [0mEnvironment info:

sys.platform                    linux
Python                          3.6.9 (default, Oct  8 2020,
   12:12:24) [GCC 8.4.0]
numpy                           1.19.5
detectron2                      0.3 @/home/appuser/detectron2_repo
   /detectron2
Compiler                        GCC 7.5
CUDA compiler                   CUDA 10.1
detectron2 arch flags           3.5, 3.7, 5.0, 5.2, 5.3, 6.0, 6.1,
    7.0, 7.5
DETECTRON2_ENV_MODULE           <not set>
PyTorch                         1.7.1+cu101 @/home/appuser/.local/
   lib/python3.6/site-packages/torch
PyTorch debug build             False
GPU available                   True
GPU 0                           Tesla V100S-PCIE-32GB (arch=7.0)
CUDA_HOME                       /usr/local/cuda
TORCH_CUDA_ARCH_LIST            Kepler;Kepler+Tesla;Maxwell;
   Maxwell+Tegra;Pascal;Volta;Turing
Pillow                          8.1.0
torchvision                     0.8.2+cu101 @/home/appuser/.local/
   lib/python3.6/site-packages/torchvision
torchvision arch flags          3.5, 5.0, 6.0, 7.0, 7.5
fvcore                          0.1.3
cv2                             3.2.0

PyTorch built with:
  - GCC 7.3
  - C++ Version: 201402
  - Intel(R) Math Kernel Library Version 2020.0.0 Product
     Build 20191122 for Intel(R) 64 architecture
     applications
  - Intel(R) MKL-DNN v1.6.0 (Git Hash 5
     ef631a030a6f73131c77892041042805a06064f)
  - OpenMP 201511 (a.k.a. OpenMP 4.5)
  - NNPACK is enabled
  - CPU capability usage: AVX2
  - CUDA Runtime 10.1
```

```
- NVCC architecture flags: -gencode;arch=compute_37,code
    =sm_37;-gencode;arch=compute_50,code=sm_50;-gencode;
    arch=compute_60,code=sm_60;-gencode;arch=compute_70,
    code=sm_70;-gencode;arch=compute_75,code=sm_75
- CuDNN 7.6.3
- Magma 2.5.2
- Build settings: BLAS=MKL, BUILD_TYPE=Release,
    CXX_FLAGS= -Wno-deprecated -fvisibility-inlines-
    hidden -DUSE_PTHREADPOOL -fopenmp -DNDEBUG -
    DUSE_FBGEMM -DUSE_QNNPACK -DUSE_PYTORCH_QNNPACK -
    DUSE_XNNPACK -DUSE_VULKAN_WRAPPER -O2 -fPIC -Wno-
    narrowing -Wall -Wextra -Werror=return-type -Wno-
    missing-field-initializers -Wno-type-limits -Wno-
    array-bounds -Wno-unknown-pragmas -Wno-sign-compare -
    Wno-unused-parameter -Wno-unused-variable -Wno-unused
    -function -Wno-unused-result -Wno-unused-local-
    typedefs -Wno-strict-overflow -Wno-strict-aliasing -
    Wno-error=deprecated-declarations -Wno-stringop-
    overflow -Wno-psabi -Wno-error=pedantic -Wno-error=
    redundant-decls -Wno-error=old-style-cast -
    fdiagnostics-color=always -faligned-new -Wno-unused-
    but-set-variable -Wno-maybe-uninitialized -fno-math-
    errno -fno-trapping-math -Werror=format -Wno-stringop
    -overflow, PERF_WITH_AVX=1, PERF_WITH_AVX2=1,
    PERF_WITH_AVX512=1, USE_CUDA=ON, USE_EXCEPTION_PTR=1,
     USE_GFLAGS=OFF, USE_GLOG=OFF, USE_MKL=ON, USE_MKLDNN
    =ON, USE_MPI=OFF, USE_NCCL=ON, USE_NNPACK=ON,
    USE_OPENMP=ON,
```

C.1. Entrenamiento de la red

Los pasos necesarios para entrenar un modelo, tanto con FPN (FewX_FPN), como sin FPN (Coco_FewX_train) son:

1. Generar la información de soporte de entrenamiento. Para ello se dispone de los *scripts* python *./datasets/coco/1_split_filter.py* y *./datasets/coco/3_gen_support_pool.py*. El primero realiza la separación de clases base y clases nuevas. El segundo genera los recortes de soporte a partir de dicha separación.

2. Modificar los archivo de configuración de la red para adaptarlos a las necesidades del experimento. Estos son *./configs/fsod/Base-FSOD-C4.yaml*

y *./configs/fsod/R_50_C4_1x.yaml*. En particular, es necesario adaptar los campos *SUPPORT_SHOT* (para ajustar el número de imágenes de soporte que se emplearan) y *DATASETS* (para ajustar el lugar de donde se extraerá la información)[1].

3. Ejecutar la red con el comando[2]:

```
CUDA_VISIBLE_DEVICES=X python3 fsod_train_net.py --num
    -gpus X --config-file configs/fsod/R_50_C4_1x.yaml
    2>&1 | tee log/fsod_train_log.txt
```

donde *CUDA_VISIBLE_DEVICES* es el id asociado a las GPUs que se van a usar y *–num-gpus*, el número de GPUs.

C.2. Entrenamientos de *finetuning*

Los pasos necesarios para hacer *finetuning* a un modelo previamente entrenado son:

1. Generar la información de soporte de las clases nuevas. Para ello es necesario:

   a) Ejecutar *generar_voc_annotations.py*. Genera un subconjunto del dataset con solo las categorías indicadas en la variable *VOC*. Es necesario ejecutarlo dos veces, la primera para generar *coco_2017_val_voc.json*; la segunda, *coco_2017_train_voc.json*.

   b) Ejecutar *genera_soporte_n_shot.py <shot> <semilla>*. Lee *coco_2017_train_voc.json* y genera *final_split_shot.json*.

   c) Ejecutar *3_gen_support_pool_test.py*. Lee *./coco/final_split_shot.json* y genera *./test_support_df.pkl* más directorio de imágenes (/support_test).

2. Adaptar el archivo de configuración del *finetuning* (*./configs/fsod/finetune_R_50_C4_1x.yaml*) a las necesidades del experimento. Especial importancia el campo *WEIGHTS*, en él se debe indicar la ruta al modelo base.

3. Ejecutar el reentrenamiento con el comando:

```
CUDA_VISIBLE_DEVICES=X python3 fsod_train_net.py --num
    -gpus X --config-file configs/fsod/
    finetune_R_50_C4_1x.yaml >> ./log/
    finetune_train_log.txt
```

donde *CUDA_VISIBLE_DEVICES* es el id asociado a las GPUs que se van a usar y *–num-gpus*, el número de GPUs.

[1]Los *datasets* están registrados en *./fewx/data/datasets/builtin.py*

[2]Las X son campos a cubrir

C.3. Evaluaciones

Para evaluar un modelo FPN (FewX_FPN) o no FPN (Coco_FewX) es necesario:

1. Generar la información de soporte de las clases nuevas:

   a) Ejecutar *generar_voc_annotations.py*. Genera un subconjunto del dataset con solo las categorías indicadas en la variable *VOC*. Es necesario ejecutarlo dos veces, la primera para generar *coco_2017_val_voc.json*; la segunda, *coco_2017_train_voc.json*.

   b) Ejecutar *genera_soporte_n_shot.py <shot> <semilla>*. Lee *coco_2017_train_voc.json* y genera *final_split_shot.json*.

   c) Ejecutar *3_gen_support_pool_test.py*. Lee *./coco/final_split_shot.json* y genera *./test_support_df.pkl* más directorio de imágenes (/support_test).

2. Ejecutar la red con los comandos:

```
CUDA_VISIBLE_DEVICES=X python3 fsod_train_net.py --num
    -gpus X --config-file configs/fsod/R_50_C4_1x.yaml
    --eval-only MODEL.WEIGHTS ./output/model_final.pth
    >> log/resultado.txt

CUDA_VISIBLE_DEVICES=X python3 fsod_train_net.py --num
    -gpus X --config-file configs/fsod/R_50_C4_1x.yaml
    --eval-only MODEL.WEIGHTS ./output/model_final.pth
    >> log/resultado.txt
```

   donde *MODEL.WEIGHTS* indica la ruta al modelo a evaluar.

C.4. Utilidades

A continuación se listan algunos *scripts* que facilitan la automatización de los experimentos:

- *experimento.sh*. Permite evaluar un modelo automáticamente para el conjunto de *shots* indicado en la variable *var*. Previamente es necesario ejcutar el paso 1a) de evaluaciones.
- *entrenar_fine.sh*. Permite hacer *finetuning* a un modelo base con diferentes *shots* y conjuntos de soporte.
- *genera_soporte_n_shot.py <shot> <semilla>*. Genera el JSON de anotaciones de soporte para un valor de *shot*.

- *generar_voc_annotations.py*. Sirve para construir un JSON de anotaciones, con solo *voc_inds* categorías, a partir de un archivo de anotaciones de coco (train o val).
- *entrenar_fine_config.py <shot> <modelo_base> <modelo_de_salida>*. Modifica el archivo de configuración de *finetuning*.
- *custom_3_gen_support_pool*. Genera el .pkl de soporte y el directorio con las imágenes, a partir del archivo JSON de soporte.

Bibliografía

[1] Qi Fan, Wei Zhuo, Chi-Keung Tang, y Yu-Wing Tai. (2020). *Few-shot object detection with attention-RPN and multi-relation detector*. En Computer Vision and Pattern Recognition (CVPR).

[2] Joseph Redmon, Santosh Divvala, Ross Girshick, y Ali Farhadi. (2016). *You Only Look Once: Unified, Real-Time Object Detection*. En Computer Vision and Pattern Recognition (CVPR).

[3] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y., Berg, A. (2016). *SSD: Single Shot MultiBox Detector*. En Lecture Notes in Computer Science, 21–37.

[4] Gongjie Zhang, Zhipeng Luo, Kaiwen Cui, y Shijian Lu. (2021). *Meta-DETR: Few-Shot Object Detection via Unified Image-Level Meta-Learning*. En arXiv preprint arXiv:2103.11731.

[5] Thomas Huang. (1996). *Computer Vision: Evolution And Promise*. En 9th CERN School of Computing. Geneva: CERN. pp. 21–25. doi:10.5170/CERN-1996-008.21. ISBN 978-9290830955

[6] Richard Szeliski. (2021). *Computer Vision: Algorithms and Applications*. En `https://szeliski.org/Book/` (visitado el 7 de mayo de 2021).

[7] Kozerawski, J., y Turk, M. (2018). *CLEAR: Cumulative LEARning for One-Shot One-Class Image Recognition*. En Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 3446–3455).

[8] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, y T. Lillicrap.(2016). *Meta-learning with memory-augmented neural networks*. En International Conference on Machine Learning (pp. 1842–1850).

[9] S. Benaim and L. Wolf. (2018). *One-shot unsupervised cross domain translation*. En Advances in Neural Information Processing Systems (pp. 2104–2114).

[10] Sergi Caelles, Kevis-Kokitsi Maninis, Jordi Pont-Tuset, Laura Leal-Taixé, Daniel Cremers, y Luc Van Gool. (2017). *One-Shot Video Object Segmentation*. En Computer Vision and Pattern Recognition (CVPR).

[11] Samaneh Azadi, Matthew Fisher, Vladimir Kim, Zhaowen Wang, Eli Shechtman, Trevor Darrell. (2017). *Multi-Content GAN for Few-Shot Font Style Transfer*. En 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 7564-7573).

[12] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, y Nando de Freitas. (2016). *Learning to learn by gradient descent by gradient descent*. En arXiv preprint arXiv:1606.04474.

[13] Sachin Ravi y Hugo Larochelle. (2017). *Optimization as a model for few-shot learning*. En International Conference on Learning Representations (ICLR).

[14] Luca Bertinetto, João F Henriques, Jack Valmadre, Philip Torr, y Andrea Vedaldi. (2016). *Learning feed-forward one-shot learners*. En Advances in Neural Information Processing Sys- tems (pp. 523–531).

[15] Trevor Hastie, Robert Tibshirani, y Jerome Friedman. (2001). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer.

[16] Yann LeCun, Patrick Haffner, Léon Bottou, y Yoshua Bengio. (1999). *Object Recognition with Gradient-Based Learning*. En Lecture Notes in Computer Science, vol 1681. Springer, Berlin, Heidelberg.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, y Jian Sun. (2015). *Deep Residual Learning for Image Recognition*. En 2016 IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 770-778).

[18] Ismail Mebsout. (2020) *Convolutional Neural Networks' mathematics*. En `https://towardsdatascience.com/convolutional-neural-networks-mathematics-1beb3e6447c0` (visitado el 9 de mayo de 2021).

[19] Vinod Nair y Geoffrey Hinton. (2010). *Rectified linear units improve restricted boltzmann machines*. En Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10). Omnipress, Madison, WI, USA, 807–814.

[20] Hossein Gholamalinezhad, Hossein Khosravi. (2020). *Pooling Methods in Deep Neural Networks, a Review*. En arXiv preprint arXiv:2009.07485.

[21] Sepp Hochreiter (1998). *The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions*. En International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 6, (pp. 107-116).

[22] Ankit Sachan. (2019) *Detailed Guide to Understand and Implement ResNets*. En `https://cv-tricks.com/keras/understand-implement-resnets/` (visitado el 10 de mayo de 2021)

[23] Ross Girshick, Jeff Donahue, Trevor Darrell, y Jitendra Malik. (2014). *Rich feature hierarchies for accurate object detection and semantic segmentation*. En Computer Vision and Pattern Recognition (CVPR).

[24] Ross Girshick. (2015). *Fast R-CNN*. En International Conference on Computer Vision (ICCV).

[25] Shaoqing Ren, Kaiming He, Ross Girshick, y Jian Sun. (2015). *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. En Conference and Workshop on Neural Information Processing Systems (NeurIPS).

[26] Jasper Uijlings, Koen Sande, Theo Gevers, y Arnold Smeulders. (2013). *Selective Search for Object Recognition*. En International Journal of Computer Vision, 104, 154-171.

[27] Yaqing Wang, Quanming Yao, James Kwok, y Lionel M. Ni. (2020). *Generalizing from a Few Examples: A Survey on Few-Shot Learning*. En arXiv preprint arXiv:1904.05046.

[28] Bingyi Kang, Zhuang Liu, Xin Wang, Fisher Yu, Jiashi Feng, y Trevor Darrell. (2019). *Few-shot Object Detection via Feature Reweighting*. En International Conference on Computer Vision (ICCV).

[29] Hao Chen, Yali Wang, Guoyou Wang, y Yu Qiao. (2018). *LSTD: A Low-Shot Transfer Detector for Object Detection*. En Association for the Advancement of Artificial Intelligence (AAAI).

[30] Xiaopeng Yan, Ziliang Chen, Anni Xu, Xiaoxi Wang, Xiaodan Liang, y Liang Lin. (2020). *Meta R-CNN : Towards General Solver for Instance-level Few-shot Learning*. En International Conference on Computer Vision (ICCV).

[31] Paperswithcode (2021).*Few-Shot Object Detection on MS-COCO (10-shot)*[Figura]. Recuperada de `https://paperswithcode.com/sota/few-shot-object-detection-on-ms-coco-10-shot` el 5 de junio de 2021.

[32] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, Serge Belongie. (2017). Feature Pyramid Networks for Object Detection. En arXiv preprint arXiv:1612.03144.

[33] Jiaxi Wu, Songtao Liu, Di Huang, Yunhong Wang. (2020). *Multi-Scale Positive Sample Refinement for Few-Shot Object Detection*. En European Conference on Computer Vision (ECCV).

[34] Xin Wang, Thomas E. Huang, Trevor Darrell, Joseph E. Gonzalez, and Fisher Yu. (2020). *Frustratingly Simple Few-Shot Object Detection*. En International Conference on Machine Learning (ICML).

[35] Yu-Xiong Wang, Deva Ramanan, and Martial Hebert. (2019). *Meta-learning to detect rare objects*. En International Conference on Computer Vision (ICCV).

[36] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, y Piotr Dollár. (2015). *Microsoft COCO: Common Objects in Context*. En arXiv preprint arXiv:1405.0312.

[37] Jia Deng, Wei Dong, Richard Socher, L. Li, Kai Li y Li Fei-Fei. (2009). *ImageNet: A large-scale hierarchical image database*. En IEEE Conference on Computer Vision and Pattern Recognition, 2009, pp. 248-255, doi: 10.1109/CVPR.2009.5206848.

[38] Alex Krizhevsky. (2009). *Learning Multiple Layers of Features from Tiny Images* cap. 3. En `https://www.cs.toronto.edu/~kriz/cifar.html` a 20 de mayo de 2021.

[39] *The PASCAL Visual Object Classes Homepage* [dataset]. Disponible en `http://host.robots.ox.ac.uk/pascal/VOC/` a 20 de mayo de 2021

[40] Tsung-Yi Lin, Michael Maire y más. *Sitio web de descarga de MS Coco dataset*. Disponible en `https://cocodataset.org/`, a 20 de mayo de 2021.

[41] Intel. *Intel® Xeon® Gold 5220 Processor*. Especificaciones consultadas en `https://ark.intel.com/content/www/us/en/ark/products/193388/intel-xeon-gold-5220-processor-24-75m-cache-2-20-ghz.html` el 5 de junio de 2021.

[42] Nvidia. *Tarjeta gráfica Nvidia Tesla V100*. Especificaciones consultadas en `https://www.nvidia.com/es-la/data-center/tesla-v100/` el 5 de junio de 2021.

[43] Dell. *Servidor para rack PowerEdge R740*. Especificaciones consultadas en `https://www.dell.com/es-es/work/shop/productdetailstxn/poweredge-r740` el 5 de junio de 2021.

[44] *Python*. Consultado en `https://www.python.org/` el 6 de junio de 2021.

[45] *Docker*. Consultado en `https://www.docker.com/` el 6 de junio de 2021.

[46] Yuxin Wu, Alexander Kirillov, Francisco Massa and Wan-Yen Lo, y Ross Girshick. (2019). *Detectron2*. Disponible en `https://github.com/facebookresearch/detectron2`

[47] *Pytroch*. Consultado en `https://pytorch.org/` el 6 de junio de 2020.

[48] *CUDA*. Consultado en `https://developer.nvidia.com/cuda-zone` el 6 de junio de 2021.

[49] *Visual Studio Code*. Consultado en `https://code.visualstudio.com/` el 6 de junio de 2021.

[50] *Overleaf*. Consultado en `https://es.overleaf.com` el 6 de junio de 2021.

[51] *Diagrams.net*. Consultado en `https://www.diagrams.net/` el 6 de junio de 2021.

[52] O'REILLY. *Detection or localization and segmentation* [Figura]. Recuperado de `https://www.oreilly.com/library/view/deep-learning-for/9781788295628/4fe36c40-7612-44b8-8846-43c0c4e64157.xhtml` el 7 de mayo de 2021.

[53] Adrian Rosebrock. *Detecting cats in images with OpenCV* [Figura]. Recuperado de `https://www.pyimagesearch.com/2016/06/20/detecting-cats-in-images-with-opencv/` el 7 de mayo de 2021.

[54] Ankit Sachan. *Detailed Guide to Understand and Implement ResNets* [Figura]. Recuperada de `https://cv-tricks.com/keras/understand-implement-resnets/` el 9 de mayo de 2021.

[55] Vineet Panchbhaiyye y Tokunbo Ogunfunmi. (2020). *A Fifo Based Accelerator for Convolutional Neural Networks* [Figura]. Recuperada de `https://www.researchgate.net/figure/Convolution-operation-in-a-CNN_fig1_341083522` el 9 de mayo de 2021.

[56] Jonathan Hui. (2018). *mAP (mean Average Precision) for Object Detection* [Figura]. Recuperada de `https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173` el 5 de junio de 2021.

[57] ADLINK. (2019). *Real-Time Object Detection Using TensorFlow* [Figura]. Recuperada de `https://goto50.ai/real-time-object-detection-using-tensorflow/` el 24 de Junio de 2021.