ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN SISTEMAS AUDIOVISUALES Y MULTIMEDIA

TRABAJO FIN DE GRADO

DEEP LEARNING EN SENSORES RGBD

Autor: David Butragueño Palomar
Tutor:

Curso académico 2017/2018

Resumen

Agradecimientos

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Contexto y motivación

Una de las fuentes de inspiración matemáticas ha sido el continuo intento de formalizar el pensamiento humano. La complejidad para simular acciones humanas frente a otras teorías matemáticas ha hecho que se separen los campos de estudio y aplicación.

Comenzando en la década de 1940 y con una gran aceleración en la década de 1980, se ha hecho un gran esfuerzo por modelar la cognición utilizando formalismos basados en modelos cada vez más sofisticados de la fisiología de las neuronas. Algunas ramas de este trabajo se siguen enfocando en la teoría biológica y psicológica, pero como ocurrió en el pasado, estos formalismos están adquiriendo una vida matemática y de aplicación propia. Mucha variedades de redes adaptativas han demostrado ser prácticas enfrentándose a problemas de gran dificultad, convirtiendo el estudio de sus propiedades matemáticas y computacionales en un campo muy interesante de estudio.

La investigación en el campo de las redes neuronales ha atraído cada vez más atención en los años recientes. Desde 1943, cuando Warren McCulloch y Walter Pitts presentaron el primer modelo de neuronas artificiales, nuevas y más sofisticadas propuestas se han hecho década tras década. El análisis matemático de estas propuestas ha resuelto algunos de los misterios planteados por los nuevos modelos pero también ha dejado muchas preguntas abiertas para futuras investigaciones. Cabe destacar que el estudio de las neuronas, sus interconexiones y su papel dentro del cerebro es uno de los campos de investigación más dinámicos e importantes en la biología moderna. Esta relevancia se puede ilustrar señalando que entre los años 1901 y

1991, aproximadamente el 10 % de los Premios Nobel de Fisiología y Medicina fueron otorgados a científicos que contribuyeron a la comprensión del cerebro. No es una exageración decir que se ha aprendido más del sistema nervioso en los últimos 50 años que nunca antes.

El objetivo final del estudio del cerebro para simular acciones humanas es implementar un sistema automático que opera sin intervención humana, en un ambiente no estructurado e incierto, es decir, que se trate de un sistema autónomo e inteligente.

Los *sistemas biológicos* ofrecen la posibilidad de diseñar *sistemas inteligentes*. Procesan información de forma no convencional, no requieren modelos de referencia y se desempeñan exitosamente en presencia de incertidumbre; aprenden a realizar nuevas tareas y se adaptan con facilidad a ambientes cambiantes.

Se dice que un sistema tiene la *capacidad de aprender* si adquiere y procesa información acerca de su desempeño y del ambiente que lo rodea, para mejorar dicho desempeño.

Para conocer el funcionamiento de los *sistemas biológicos*, y su posterior uso para la creación de *sistemas inteligentes artificiales*, es necesario entender lo que es una neurona. Una neurona es el componente básico del sistema nervioso. Se tratan de células divididas en 3 partes principales: las dendrinas, el cuerpo de la célula o soma y el axón. Las dendrinas son el elemento receptor, encargadas de proporcionar las señales eléctricas al cuerpo de la célula; el soma realiza la suma de todas esas señales y el axón es una fibra larga que lleva la señal desde el cuerpo de la célula hacia otras. El punto de contacto entre el axón de una célula y la dendrina de otra se denomina sinapsis.

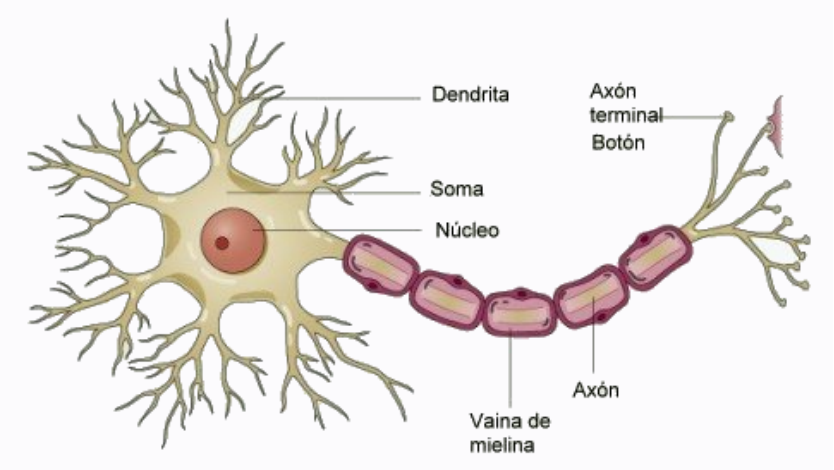

Figura 1.1: Partes de una neurona biológica

Este punto de unión es el encargado de transferir los impulsos eléctricos generados en la membrana de la neurona. Esta información viaja a través de los axones en breves impulsos eléctricos.

El conjunto de un gran número de neuronas conectadas entre sí forman la llamada red neuronal. Conforman el sistema nervioso y el cerebro: el cerebro humano puede contener 10^{11} neuronas y 10^{15} interconexiones. Las redes neuronales biológicas pueden establecerse como grupos de neuronas activas especializadas en tareas como cálculos matemáticos, posicionamiento y memoria.

A partir de aquí, se crean las redes neuronales artificiales las cuales son modelos simplificados de las redes neuronales biológicas. Tratan de extraer las capacidades del cerebro para resolver ciertos problemas complejos como reconocimiento de patrones o control moto-sensorial.

Consisten en un gran número de elementos simples de procesamiento llamados nodos o neuronas que están organizados en capas. Cada neurona está conectada con otras neuronas mediante enlaces de comunicación, cada uno de los cuales tiene asociado un peso. Los pesos representan la información que será usada por la red neuronal para resolver un problema determinado.

Así, las redes neuronales artificiales son sistemas adaptativos que aprenden de la experiencia, es decir, aprenden a llevar a cabo ciertas tareas mediante un entrenamiento con ejemplos ilustrativos. Mediante este entrenamiento

o aprendizaje, las redes neuronales artificiales crean su propia representación interna del problema. Posteriormente, pueden responder adecuadamente cuando se les presentan situaciones a las que no habían sido expuestas anteriormente, es decir, las redes neuronales artificiales son capaces de generalizar de casos anteriores a casos nuevos.

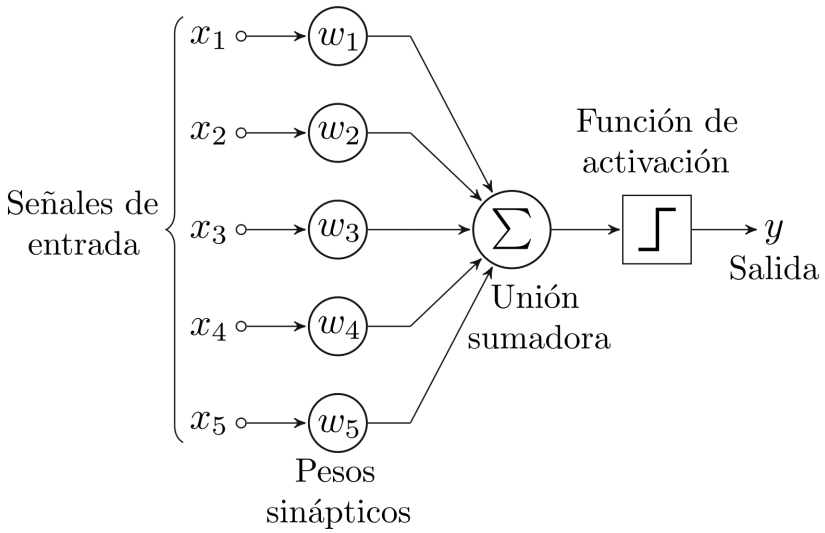

Figura 1.2: Modelo de una red neuronal artificial

En el modelo de una red neuronal, representado en la figura 1.2, se pueden identificar varios elementos.

- **Enlaces de conexión:** Parametrizados por los pesos sinápticos w_n. Si w_n ¿0 la conexión es excitadora mientras que si w_n ¡0, la conexión será inhibidora.
- **Sumador:** Suma los componentes de las señales de entrada multiplicadas por los pesos sinápticos w_n.
- **Función de activación:** Transformación no lineal del sumatorio obtenido.

Como se ha comentado anteriormente, el conocimiento de una red neuronal se encuentra distribuido en los pesos de las conexiones existentes entre todas las conexiones de la red. La red neuronal aprende variando el valor

de sus parámetros, en particular los pesos sinápticos y el umbral de polarización. El aprendizaje es un proceso por el cual los parámetros se adaptan por la iteración continua con el medio ambiente. El tipo de aprendizaje está determinado por la forma en que se realiza dicha adaptación. Este proceso implica la siguiente secuencia de eventos:

- La red neuronal es estimulada por el medio ambiente.
- La red neuronal ajusta sus parámetros
- La red neuronal genera una nueva respuesta.

Actualmente, existen varios criterios, llamados de forma genérica reglas de aprendizaje, para modificar los pesos de la red y así conseguir que aprenda a solucionar un determinado problema. Las reglas de aprendizaje consisten generalmente en algoritmos matemáticos que pueden llegar a ser sumamente complejos. Se suelen considerar dos tipos de reglas de aprendizaje: aprendizaje supervisado y aprendizaje no supervisado.

- En el aprendizaje supervisado existe un supervisor que controla el proceso de aprendizaje de la red. El supervisor comprueba la salida de la red en respuesta a una determinada entrada y en el caso de que la salida no coincida con la deseada, se procede a modificar los pesos de las conexiones, con el fin de conseguir que la salida obtenida se aproxime a la deseada.
- Con el aprendizaje no supervisado, también llamado autoorganizado, la red no requiere influencia de un supervisor para ajustar los pesos de las conexiones entre sus neuronas. La red no recibe ninguna información por parte del entorno que le indique si salida generada en respuesta a una determinada entrada es o no correcta. Su función consiste en encontrar las características, regularidades o categorías que se puedan establecer entre los datos que se presentan en su entrada.

Una vez obtenidos y guardados los pesos óptimos en la fase de entrenamiento, es necesario medir la eficacia de la red de forma objetiva mediante la presentación de casos nuevos, es decir, entradas diferentes a los casos de entrenamiento, de forma que a la fase de entrenamiento le debe seguir una fase de test. En esta fase no se modifican los pesos, simplemente se presentan casos nuevos, llamados casos de test, a la entrada de la red y ésta proporciona

una salida para cada uno de ellos. Si se comprueba que se siguen obteniendo resultados dentro del margen de error deseado, se puede proceder a emplear la red neuronal dentro de su entorno de trabajo real.

Capítulo 2

Infraestructura

2.1. Caffe

Caffe es un framework de aprendizaje profundo desarrollado por Berkeley AI Research (BAIR) y por contribuyentes de la comunidad. Fue creado por Yangqing Jia durante su doctorado en UC Berkeley. Caffe está publicado bajo la licencia BSD 2-Clause.

2.1.1. Línea de comandos

Caffe dispone de una interfaz de línea de comandos llamada *cmdcaffe* la cual es la herramienta utilizada por Caffe para el entrenamiento del modelo y el diagnóstico del mismo. Los principales comandos que se pueden ejecutar son:

- **Entrenamiento:** Con el comando *caffe train* es posible aprender modelos desde cero,
- **Test:** El comando *caffe test* puntúa los modelos ejecutándolos en la fase de test e informa de la salida de la red como su puntuación. En primer lugar se informa la puntuación por lotes de datos de entrada y finalmente el promedio general.
- **Comparación:** El comando *caffe time* compara el modelo capa a capa. Esto es util para comprobar el rendimiento del sistema y medir los tiempos de ejecución relativos a los modelos.

2.1.2. Python

La interfaz de Python *pycaffe* contiene el módulo de Caffe y sus propios scripts en la ruta caffe/python. Con el comando *import caffe* se importará esta interfaz, pudiendo así cargar diferentes modelos de Caffe, manejar instrucciones de entrada/salida, visualizar redes y numerosas funcionalidades más. Todos los datos y parámetros se encuentran disponibles tanto para lectura como para escritura. Algunas de las tareas que se pueden realizar con está interfaz son:

- **caffe.Net** es la interfaz central para cargar, configurar y ejecutar modelos.
- **caffe.Classifier y caffe.Detector** proporcionan interfaces para tareas de clasificación y detección.
- **caffe.SGDSolver** se trata de la interfaz de resolución.
- **caffe.io** maneja funciones de entrada/salida con preprocesamiento.
- **caffe.draw** visualiza las arquitecturas de red.

2.1.3. Capas de una red neuronal

Para crear un modelo de Caffe es necesario definir la arquitectura del mismo utilizando para ello un archivo de definición de buffer de protocolo (prototxt).

Capas de datos

Los datos entran en Caffe a través de las capas de datos las cuáles se encuentran en la parte inferior de las redes. Estos datos pueden provenir de bases de datos (LevelDB o LMDB), directamente de la memoria, o, cuando la eficiencia no es crítica, desde archivos en disco en formato HDF5 o formatos de imagen comunes.

Tareas comunes de preprocesamiento de los datos de entrada, tales como escalado o reflejo, están disponibles especificando *TransformationParameters* por algunas de las capas. Los tipos de capas "bias", "scalez çrop"pueden ser útiles para el preprocesamiento de la entrada cuando la opción *TransformationParameters* no está disponible.

- **Image Data:** Lee imágenes sin procesar.
- **Database:** Lee los datos de LevelLDB o LMDB.
- **HDF5 Input:** Lee los datos en formato HDF5 permitiendo que estos tengan dimensiones arbitrarias.
- **HDF5 Output:** Escribe datos en formato HDF5
- **Input:** Normalmente utilizada para redes que se están implementando.
- **Window Data:**
- **Memory Data:** Lee archivos directamente desde memoria.
- **Dummy Data:** Utilizado para datos estáticos.

Capas de visión

Las capas de visión, generalmente toman imágenes como datos de entradas y generan otras imágenes como salida aunque también pueden tomar datos de otros tipos y dimensiones. Una imagen puede tener un canal ($c = 1$) si se trata de una imagen en escala de grises o 3 canales ($c = 3$) si se trata de una imagen RGB. Pero en este contexto, las características distintivas para el tratamiento de las imágenes de entrada serán la altura y la anchura de las mismas. La mayoría de las capas de visión trabajan aplicando una operación particular sobre alguna región de la entrada para producir una región correspondiente a la salida.

- **Convolution Layer:** Convoluciona la imagen de entrada con un conjunto de filtros. Este proceso transforma la matriz de píxeles de la imagen original en otra matriz de características. Esta nueva imagen es esencialmente la original pero cada píxel tiene mucha más información ya que contiene información de la región en la que se encuentra el píxel, no sólo del píxel aislado.

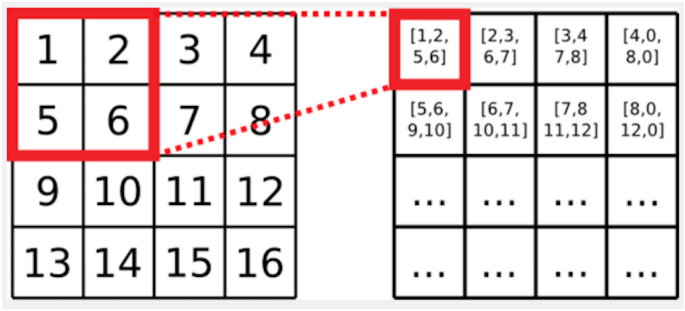

Figura 2.1: Ejemplo convolución con tamaño del núcleo 2x2

- **Pooling Layer:** Realiza *pooling* de los datos de entrada utilizando para ello funciones de máximo, media o estocásticas. La capa de pooling se coloca generalmente después de la capa de convolución. Su utilidad principal radica en la reducción espacial de la imagen de entrada. Para ello, se divide el mapa de características obtenido anteriormente en un conjunto de bloques de m x n. A continuación, se aplica una función de agrupación para cada uno de los bloques. Tras este proceso, se obtendrá una matriz de características más pequeño. Dentro de las funciones de agrupación, destacan max pooling, el cual elige el valor más alto dentro del bloque y pooling promedio (average) el cual toma como respuesta de bloque el valor promedio de las respuestas del bloque.

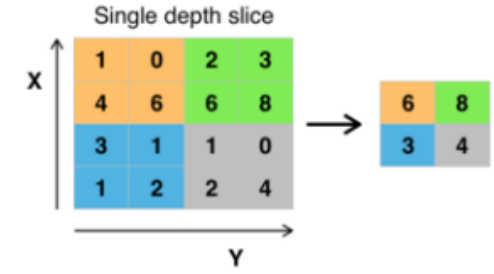

Figura 2.2: Pooling con función de agrupación del máximo

- **Spatial Pyramid Pooling (SPP)**
- **Crop**

- **Deconvolution Layer:** Realiza uno convolución transpuesta.
- **Im2Col**

Capas recurrentes

- **Recurrent**
- **RRNN**
- **Long-Short Term Memory (LSTM)**

Capas comunes

- **Inner Product:** Capa totalmente conectada
- **Dropout**
- **Embed**

Capas de pérdida

Estas capas de pérdida conducen al aprendizaje comparando la salida obtenida con el valor de la entrada asignado así un coste para minimizarla.

- **Multinomial Logistic Loss**
- **Infogain Loss**
- **Softmax with Loss**
- **Sum-of-Squares / Euclidean**
- **Hinge / Margin**
- **Sigmoid Cross-Entropy Loss**
- **Accuracy / Top-k layer**
- **Contrastive Loss**

Capas de normalización

- **Local Response Normalization (LRN):** Normaliza regiones locales de los datos de entrada.
- **Mean Variance Normalization (MVN):** Realiza una normalización de contraste / normalización de instancia.
- **Batch Normalization:** Realiza normalizaciones sobre pequeños lotes de datos de entrada.

Capas de activación

En general, estas capas son operados que toman un dato de la salida de la capa anterior y generan datos con las mismas dimensiones.

- **ReLU / Rectified-Linear and Leaky-ReLU:** Se trata de una función lineal, rectilínea con pendiente uniforme.

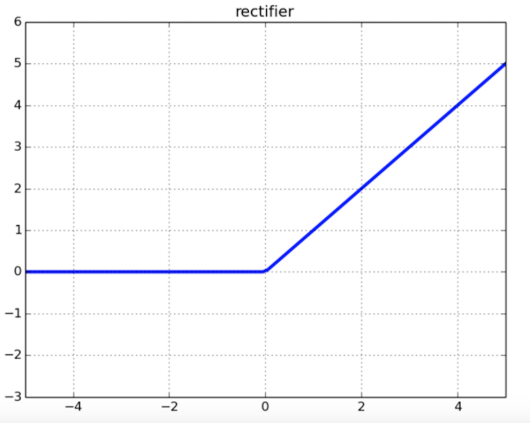

Figura 2.3: Función de activación ReLu

Se puede definir a partir de la siguiente ecuación:

$$f(x) = \begin{cases} 0 \text{ si } x < 0 \\ \text{x si } x \geq 0 \end{cases} \tag{2.1}$$

- **PReLU**
- **ELU**
- **Sigmoid**

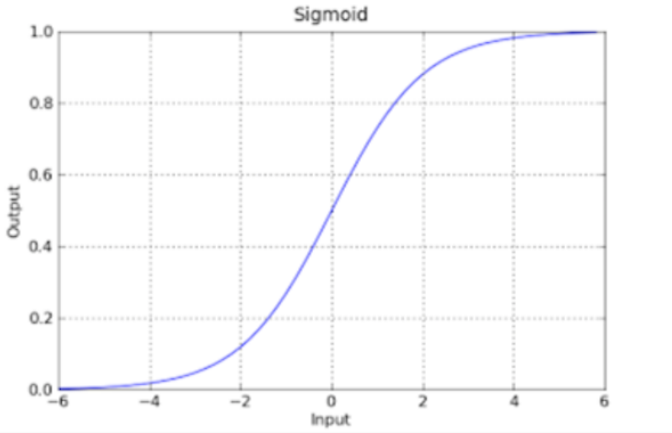

Figura 2.4: Función de activación Sigmoide

Se puede definir a partir de la siguiente ecuación:

$$f(x) = \frac{1}{1 + e^{-x}}$$

- **TanH**
- **Absolute Value**
- **Power**

$$f(x) = (shift + scale * x)^{power} \tag{2.2}$$

- **Exp**

$$f(x) = base^{(shift+scale*x)} \tag{2.3}$$

- **Log**

$$f(x) = log(x) \tag{2.4}$$

- **BNLL**

$$f(x) = log(1 + exp(x)) \tag{2.5}$$

- **Threshold**: Realiza la función de paso en el umbral definido por el usuario.
- **Bias**
- **Scale**

2.2. JdeRobot

Se trata de un framework cuyo objetivo es desarrollar aplicaciones en robótica y visión por computadora. También tiene actuación en domótica y en escenarios con sensores, accionadores y software inteligente. Ha sido desarrollado para ayudar en la programación de este software inteligente. Está escrito principalmente utilizando el lenguaje C++ proporcionando un entorno de programación en el que el programa de aplicación está compuesto por una colección de varios componentes asíncronos concurrentes. Estos componentes pueden ejecutarse en diferentes equipos y están conectados mediante el middleware de comunicaciones ICE. Los componentes pueden estar escritos en C++, Python, Java y todos ellos ellos interactúan a través de interfaces ICE explícitas.

JdeRobot simplifica el acceso a dispositivos hardware desde el programa de control. Obtener mediciones de sensores es tan simple como llamar a una función local y ordenar comandos de motor tan fácil como llamar a otra función local. La plataforma adjunta esas llamadas a invocaciones remotas sobre los componentes conectados al sensor o los dispositivos de accionamiento. También, pueden conectarse a sensores y activadores reales o simulados, tanto a nivel local como remoto utilizando para ello la red. Esas funciones construyen la API para la capa de abstracción del hardware. La aplicación robótica obtiene las lecturas del sensor y ordena los comandos del actuador usando esa API para desplegar su comportamiento. Se han desarrollado varios drivers para soportar diferentes sensores, activadores y simuladores. Los robots y sensores actualmente soportados son:

- **Sensores RGBD:** Kinect and Kinect2 de Microsoft, Asus Xtion
- **Robots con ruedas:** Kobuki (TurtleBot) de Yujin Robot y Pioneer de MobileRobotics Inc.
- **ArDrone quadrotor de Parrot**
- **Escáneres laser:** LMS de SICK, URG de Hokuyo y RPLidar
- **Simulador Gazebo**
- **Cámaras Firewire, cámaras USB, archivos de vídeo (mpeg, avi), cámaras IP (como Axis)**

JdeRobot incluye varias herramientas de programación de robots y bibliotecas. En primer lugar, teleespectadores y teleoperadores para varios robots y sus sensores y motores. En segundo lugar, un componente de calibración de cámara y una herramienta de tunning para filtros de color. En tercer lugar, una herramienta llamada VisualHFSM para la programación del comportamiento del robot utilizando la jerarquía Finite State Machines. Además, también proporciona una biblioteca para desarrollar controladores difusos y otra para la geometría proyectiva y el procesamiento de la visión por computadora.

Cada componente puede tener su propia interfaz gráfica de usuario o ninguna en absoluto. Actualmente, las bibliotecas GTK y Qt son compatibles, incluyéndose varios ejemplos de OpenGL para gráficos 3D con ambas bibliotecas.

JdeRobot es un software de código abierto con licencia como GPL y LGPL. También utiliza software de terceros como el simulador Gazebo, ROS, OpenGL, GTK, Qt, Player, Stage, GSL, OpenCV, PCL, Eigen u Ogre.

2.2.1. CameraServer y CameraView

2.3. Formato de ficheros

2.3.1. eXtensible Markup Language

XML (eXtensible Markup Language - Lenguaje de Marcas Extensible) es un meta-lenguaje que permite definir lenguajes de marcas, desarrollado por *World Wide Web Consortium* (W3C) y utilizado para almacenar datos de forma legible. A diferencia de otros lenguajes, XML da soporte a bases

de datos, siendo útil cuando varias aplicaciones deben comunicarse entre sí o integrar información.

XML presenta varias ventajas frente a otros lenguajes, las cuales se definen a continuación:

- Es extensible, es decir, después de ser diseñado y puesto en producción, es posible la adición de nuevas etiquetas.
- Mejora la compatibilidad entre aplicaciones. Se pueden comunicar aplicaciones de distintas plataformas sin que importe el origen de los datos.
- Se transforman datos en información, pues se les añade un significado concreto y y se les asocia a un contexto, con lo cual existe flexibilidad para estructurar documentos.

La tecnología XML busca dar solución el problema de expresar información estructurada de la manera más abstracta y reutilizable posible. Que la información sea estructurada quiere decir que se compone de partes bien definidas, y que esas partes, a su vez, se componen de otras partes. De esta forma, surge un árbol de partes de información. Estas partes se llaman *elementos* y vienen definidas mediante *etiquetas*.

Una etiqueta consiste en una marca hecha en el documento, que señala una porción de éste como un elemento. Se trata de información con un sentido claro y definido. Las etiquetas tienen la forma ¡nombre¿, donde *nombre* es el nombre del elemento que se está señalando.

En la sección 2.4.1 se puede observar un ejemplo de la estructura de un archivo XML.

2.3.2. JavaScript Object Notation

JSON (JavaScript Object Notation - Notación de Objetos de JavaScript) es un formato basado en texto estándar para representar datos estructurados en la sintaxis de objetos de JavaScript. Es comúnmente utilizado para transmitir datos en aplicaciones web, como por ejemplo enviar datos desde el servidor al cliente, haciendo posible que esos datos puedan ser mostrados en páginas web. Aunque es muy parecido a la sintaxis de objeto literal de JavaScript, puede ser utilizado independientemente de éste, y muchos ambientes de

programación poseen la capacidad analizar, parsear y generar ficheros JSON.

La tecnología JSON tiene varios tipos de datos disponibles, los cuales se presentan a continuación:

- **Números:** Se permiten números negativos y opcionalmente pueden contener parte fraccional separada por puntos.
- **Cadenas:** Representan secuencias de 0 o más caracteres. Se ponen entre dobles comillas.
- **Booleanos:** Se permite la representación de valores booleanos.
- **Null:** Representa un valor nulo.
- **Array:** Representa una lista ordenada de cero o más valores que pueden ser de cualquier tipo. Los valores se separan por comas y el vector se mete entre corchetes.
- **Objetos:** Son colecciones no ordenadas de pares *clave:valor* separados por comas y dispuestas entre llaves. La clave tiene que ser una cadena mientras que el valor puede ser de cualquier tipo.

En la sección 2.4.2 se puede observar un ejemplo de la estructura de un archivo JSON.

2.3.3. Lightning Memory-Mapped Database

LMDB (Lightning Memory-Mapped Database) es una biblioteca de gestión de bases de datos modelada en la API de BerkeleyDB. Toda la base de datos está modelada como un mapa de memoria, y todos los datos recuperados de ella son datos devueltos directamente de la memoria asignada.

La biblioteca es totalmente compatible con subprocesos y admite el acceso simultáneo de lectura y escritura desde múltiples procesos o hilos. Las páginas de datos utilizan una estrategia que permite que no se sobreescriban las páginas con datos activos, lo cual brinda además resistencia a la corrupción de la base de datos. Las escrituras están totalmente serializadas, es decir, solo se permite una escritura de forma simultánea. La estructura de la base de datos tiene múltiples versiones para que las lecturas se ejecuten sin bloqueos; las escrituras sobre ésta no bloquean a las lecturas, y las lecturas no bloquean las escrituras.

2.4. Bases de datos

2.4.1. PascalVOC

Esta base de datos contiene **11.530** imágenes de entrenamiento y validación que representan **27.450** objetos diferentes distribuidos en **20 clases**. Los datos de entrenamiento proporcionados consisten en un conjunto de imágenes; cada imagen tiene un archivo de anotación que proporciona un cuadro delimitador o *bounding box* y una etiqueta de las 20 clases para cada objeto presente en la imagen. Por lo tanto, la tarea de detección consiste en predecir el cuadro delimitador y la etiqueta de cada objeto en la imagen de prueba.

Es importante saber cuántas imágenes aparecen en cada uno de los 20 objetos para saber si la base de datos está bien escalada, o si, por el contrario, algunos objetos aparecen con más frecuencia que otros. En la base de datos VOC2012, las distribuciones de imágenes y objetos por clase son aproximadamente iguales en todos los conjuntos de entrenamiento / validación y prueba. Específicamente, la distribución de objetos para la tarea de detección se muestra en la siguiente imagen.

	train		val		trainval		test	
	Images	Objects	Images	Objects	Images	Objects	Images	Objects
Aeroplane	327	432	343	433	670	865	-	-
Bicycle	268	353	284	358	552	711	-	-
Bird	395	560	370	559	765	1119	-	-
Boat	260	426	248	424	508	850	-	-
Bottle	365	629	341	630	706	1259	-	-
Bus	213	292	208	301	421	593	-	-
Car	590	1013	571	1004	1161	2017	-	-
Cat	539	605	541	612	1080	1217	-	-
Chair	566	1178	553	1176	1119	2354	-	-
Cow	151	290	152	298	303	588	-	-
Diningtable	269	304	269	305	538	609	-	-
Dog	632	756	654	759	1286	1515	-	-
Horse	237	350	245	360	482	710	-	-
Motorbike	265	357	261	356	526	713	-	-
Person	1994	4194	2093	4372	4087	8566	-	-
Pottedplant	269	484	258	489	527	973	-	-
Sheep	171	400	154	413	325	813	-	-
Sofa	257	281	250	285	507	566	-	-
Train	273	313	271	315	544	628	-	-
Tvmonitor	290	392	285	392	575	784	-	-
Total	5717	13609	5823	13841	11540	27450	-	-

Figura 2.5: Distribución de objetos en PascalVOC

En cuanto a los archivos de anotaciones, estos tienen un formato XML. En primer lugar, aparecen algunas etiquetas informativas acerca de la imagen, como la etiqueta *filename* que indica el nombre de la imagen, o la etiqueta *database* que indica el nombre de la base de datos.

```
<folder>VOC2012</folder>
<filename>2007_009084.jpg</filename>
<source>
    <database>The VOC2007 Database</database>
    <annotation>PASCAL VOC2007</annotation>
    <image>flickr</image>
</source>
```

Después, aparece el grupo de etiquetas *size*, donde se indican el ancho de la imagen, *width*, el alto, *height* y la profundidad, *depth*.

```
<size>
```

```
    <width>500</width>
    <height>375</height>
    <depth>3</depth>
</size>
```

Finalmente, aparecen las etiquetas referentes a cada uno de los objetos existentes en la imagen. Destacan la etiqueta *name*, la cual indica el nombre de la etiqueta del objeto, y el grupo *bndbox*, que representa la *bounding box* a partir de las coordenadas *x_min*, *y_min*, *x_max* y *y_max*.

```
<object>
    <name>motorbike</name>
    <pose>Right</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
        <xmin>43</xmin>
        <ymin>77</ymin>
        <xmax>481</xmax>
        <ymax>375</ymax>
    </bndbox>
</object>
```

2.4.2. COCO

COCO es un conjunto de datos creado para la detección y segmentación de objetos y para generación de subtítulos a gran escala. Algunas de las características de esta base de datos son:

- Más de 300.000 imágenes
- 1.5 millones de instancias de objetos
- 80 categorias de objetos

Actualmente, COCO utiliza 3 tipos de anotaciones: instancias de objetos, puntos claves de objetos y leyendas de imágenes. Las anotaciones se almacenan usando el formato de archivo JSON. Todas las anotaciones comparten la estructura de datos básica definida a continuación:

```
{
   "info"              : info,
   "images"            : [image],
   "annotations"       : [annotation],
   "licenses"          : [license],
}

info{
   "year"              : int,
   "version"           : str,
   "description"       : str,
   "contributor"       : str,
   "url"               : str,
   "date_created"      : datetime,
}

image{
   "id"                : int,
   "width"             : int,
   "height"            : int,
   "file_name"         : str,
   "license"           : int,
   "flickr_url"        : str,
   "coco_url"          : str,
   "date_captured"     : datetime,
}

license{
   "id"                : int,
   "name"              : str,
   "url"               : str,
}
```

Figura 2.6: Estructura básica de anotaciones en COCO

Para la tarea de detección, es de especial interés la anotación usando instancias de objetos. Cada anotación de instancia contiene una serie de campos, incluida la identificación de categoría y la máscara de segmentación del objeto. El formato de segmentación depende de si la instancia representa un solo objeto (iscrowd = 0 en cuyo caso se usan polígonos) o una colección de objetos (iscrowd = 1 en cuyo caso se usa RLE). Hay que tener en cuenta que un solo objeto (iscrowd = 0) puede requerir múltiples polígonos. Las anotaciones de multitudes (iscrowd = 1) se utilizan para etiquetar grandes grupos de objetos (por ejemplo, una multitud de personas). Además, se proporciona un cuadro delimitador para cada objeto (las coordenadas del cuadro se miden desde la esquina superior izquierda de la imagen y están indexadas en 0). Finalmente, el campo de categorías de la estructura de anotación almacena la asignación de los nombres de categoría y supercategoría.

```
annotation{
   "id"                : int,
   "image_id"          : int,
   "category_id"       : int,
   "segmentation"      : RLE or [polygon],
   "area"              : float,
   "bbox"              : [x,y,width,height],
   "iscrowd"           : 0 or 1,
}

categories[{
   "id"                : int,
   "name"              : str,
   "supercategory"     : str,
}]
```

Figura 2.7: Estructura básica de anotaciones en COCO

En referencia a los datos, el conjunto de datos MS COCO se divide en dos partes aproximadamente iguales. La primera mitad del conjunto de datos se lanzó en 2014, mientras que la segunda mitad se lanzó en 2015. La versión 2014 contiene **82.783** imágenes para el entrenamiento, **40.504** para validaciones y **40.775** imágenes de prueba (aproximadamente 1/2 de imágenes de entrenamiento, 1/4 de validación y 1/4 de prueba). Hay casi 270 mil personas segmentadas y un total de 886 mil instancias de objetos segmentados en los datos de entrenamiento y validación. La versión acumulada de 2015 contendrá un total de **165.482** imágenes de entrenamiento, **81.208** de validación y **81.434** de prueba.

La distribución de los objetos en esta base de datos se puede obtener desde su sitio web. En la sección *Explorar*, es posible elegir y combinar cada uno de los objetos y observar cuántas imágenes aparecen. La distribución de cada uno de los objetos en el conjunto de entrenamiento / validación se muestra en la siguiente imagen:

	images		images		images		images
person	66808	cat	4298	wine glass	2643	dinning table	12338
backpack	5756	dog	4562	cup	9579	toilet	3502
umbrella	4142	horse	3069	fork	3710	tv	4768
handbag	7133	sheep	1594	knife	4507	laptop	3707
tie	3955	cow	2055	spoon	3682	mouse	1964
suitcase	2507	elephant	2232	bowl	7425	remote	3221
bicycle	3401	bear	1009	banana	2346	keyboard	2221
car	12786	zebra	2001	apple	1662	cell phone	5017
motorcycle	3661	giraffe	2647	sandwich	2463	microwave	1601
airplane	3083	frisbee	2268	orange	1784	oven	2992
bus	4141	skis	3202	broccoli	2010	toaster	225
train	3745	snowboard	1703	carrot	1764	sink	4865
truck	6377	sports ball	4431	hot dog	1273	refrigerator	2461
boat	3146	kite	2352	pizza	3319	book	5562
traffic light	4330	baseball bat	2603	donut	1585	clock	4863
fire hydrant	1797	baseball glove	2729	cake	3049	vase	3730
stop sign	1803	skateboard	3603	chair	13354	scissors	975
parking meter	742	surfboard	3635	couch	4618	teddy bear	2234
bench	5805	tennis racket	3561	potted plant	4624	hair drier	198
bird	3362	bottle	8880	bed	3831	toothbrush	1041

Figura 2.8: Distribución de objetos en COCO

Capítulo 3

Detección

3.1. Técnica SSD

Single Shot MultiBox Detector (SSD) es un framework diseñado para detectar objetos en imágenes utilizando para ello únicamente una red neuronal profunda. Fue lanzado a finales de noviembre de 2016 y obtuvo resultados muy positivos en términos de rendimiento y precisión en tareas de detección de objetos obteniendo más del 74 % en *mAP* (mean Average Precision) a 59 *frames* por segundo en conjuntos de datos estándar como PascalVOC y COCO. Para una mejor comprensión de esta técnica, se puede comenzar con definir el significado de su nombre:

- **Single Shot:** Hace referencia a que la tareas de localización y clasificación de objetos se realizan en un único paso hacia adelante de la red.
- **MultiBox:** Es el nombre de una técnica para regresión de *bounding box* desarrollada por Szegedy.
- **Detector:** La red neuronal se trata de un detector de objetos que también clasifica esos objetos detectados.

En el momento de la predicción, la red neuronal genera las puntuaciones para cada categoría de objetos en cada cuadro predeterminado y produce ajustes en este cuadro para ajustarse mejor a la forma del objeto. Además, la red combina predicciones de múltiples mapas de características con diferentes resoluciones para manejar de forma natural objetos de varios tamaños.

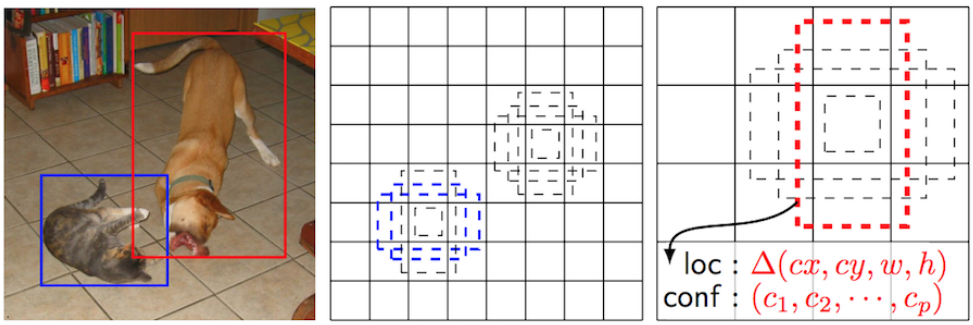

Figura 3.1: Framework SSD

(a) SSD solo necesita una imagen de entrada y los cuadros delimitadores reales para cada objeto durante el entrenamiento. De una manera convolucional, se evalúa un pequeño conjunto de cuadros predeterminados (p.ej. 4) de diferentes relaciones de aspecto para cada ubicación de éstos en varios mapas de características a diferentes escalas (p. ej 8 x 8 y 4 x 4 en (b) y (c). Para cada cuadro predeterminado se predicen tanto los desplazamiento de forma como la confianza para todas las categorías de objetos (*(c1, c2, ..., cp)*). En el momento del entrenamiento, se comparan estos cuadros predeterminados con los cuadros reales.

3.1.1. Arquitectura

El modelo SSD se basa en una red convolucional que produce un conjunto de recuadros delimitadores o *bounding boxes* de tamaño fijo y puntuaciones para la presencia de instancias de clases de objetos dentro de esos cuadros, seguido por un paso de supresión no máxima para producir las detecciones finales. Las primeras capas de la red se basan en una arquitectura VGG-16 pero descartando las capas totalmente conectadas de ésta. La razón por la que se usa VGG-16 es su gran desempeño en la clasificación de imágenes de alta calidad. En lugar de las capas totalmente conectadas de VGG, se agregaron un conjunto de capas convolucionales auxiliares, lo que permite extraer características en múltiples escalas y disminuir progresivamente el tamaño de la imagen de entrada. Posteriormente, se agrega una estructura auxiliar a la red para producir detecciones. Esta estructura tiene las siguientes características claves:

- **Mapas de características con múltiples escalas para la detección:** Se agregan varias capas de convolucionales al final de la estructu-

ra estándar comentada anteriormente. Estas capas disminuyen de forma progresiva el tamaño de la imagen de entrada y permiten predecir detecciones a múltiples escalas.

- **Predictores convolucionales para la detección:** Cada capa de características añadida puede producir un conjunto fijo de predicciones de detección utilizando para ello un conjunto de filtros convolucionales. Estos filtros están indicados en la parte superior de la arquitectura de la red SSD en la figura 3.2. Para una capa de características de tamaño m X n con p canales, el elemento básico para predecir los parámetros de una posible detección es un núcleo pequeño de 3 x 3 x p el cuál produce una puntuación para una categoría determinada. En cada una de las posiciones por donde se aplica el filtro, éste produce un valor de salida.

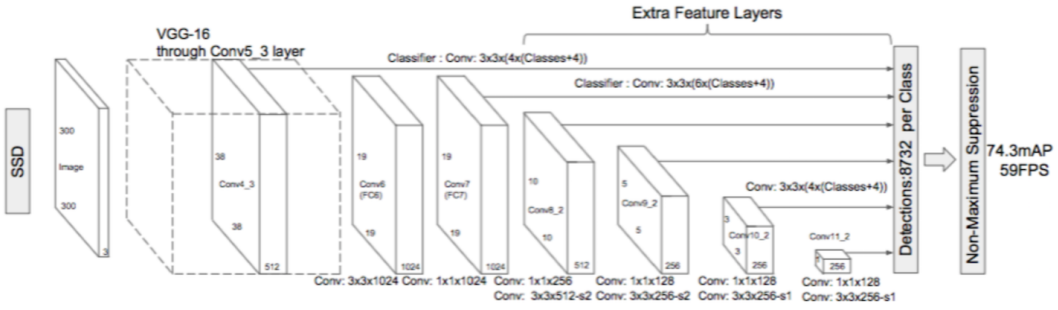

Figura 3.2: Arquitectura SSD

- **Cajas predeterminadas y relaciones de aspecto:** Se asocian un conjunto de cuadros delimitadores por defecto con cada celda del mapa de características. Los cuadros predeterminados marcan el mapa de características de un manera convolucional, por lo que la posición de cada cuadro con respecto a su celda es fija. En cada celda del mapa de características, se predicen los desplazamientos relativos a las formas de los cuadros predeterminados en cada una de las celdas, así como la puntuación por clase que indica la presencia de una instancia de clase en cada uno de los cuadros delimitadores. Específicamente, para cada cuadro k en una posición determinada se calcula la puntuación de clase c y los 4 desplazamientos relativos a la forma del cuadro por defecto original. Como resultado se obtienen $(c + 4)k$ filtros que se aplican alrededor de cada ubicación en el mapa de características, produciendo

(c + 4)kmn de salidas para un mapa de *m X n*. Se permiten diferentes formas para los cuadros predeterminados en los mapas de características con el objetivo de discretizar el espacio para las posibles formas que tengan las cuadros delimitadores de salida.

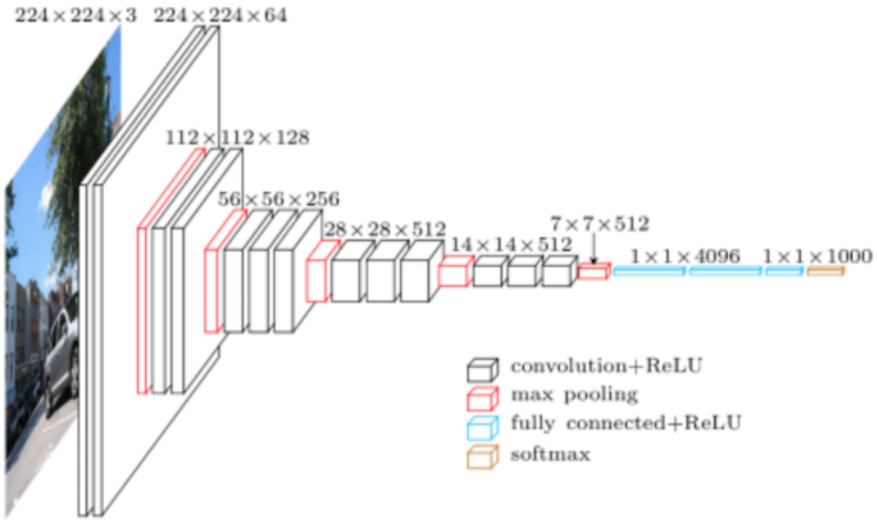

Figura 3.3: Arquitectura VGG-16

3.1.2. Índice Jaccard

El índice Jaccard, también conocido como **Intersection over Union**, es una estadística utilizada para comparar la similitud y la diversidad de conjuntos de muestras. Este índice mide la similitud entre conjuntos de muestras finitas y se define como el tamaño de la intersección dividido por el tamaño de la unión de los conjuntos de muestras.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{3.1}$$

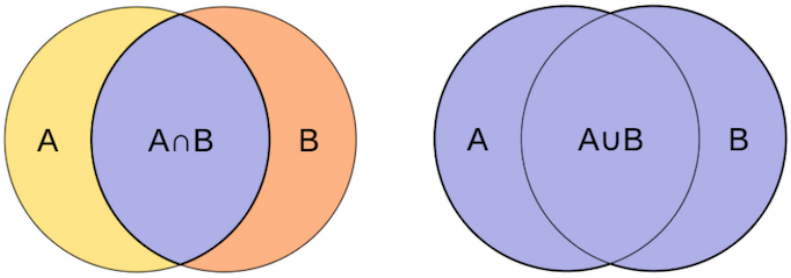

Figura 3.4: Intersección y unión sobre 2 conjuntos A y B

Si no hay coincidencias en ambos conjuntos, el índice Jaccard se definirá como J(A,B) = 1.

$$0 \leq J(A, B) \leq 1 \tag{3.2}$$

Por lo tanto, el índice Jaccard o **Intersection over Union** (IoU) se puede definir gráficamente de la siguiente manera:

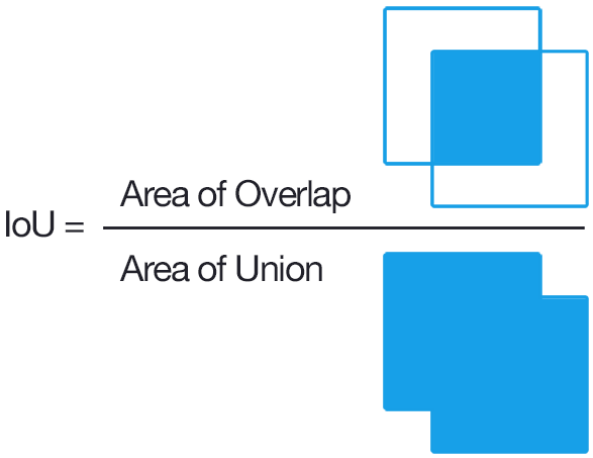

Figura 3.5: Intersection over Union

Para determinar la calidad de un detector, se utilizarán la *bounding box* real, obtenida de las anotaciones, y la *bounding box* detectada, calculando de esta manera el índice Jaccard. Por lo tanto, podemos evaluar la calidad de un detector a partir del valor obtenido:

- Si el índice Jaccard tiene un valor alrededor de **0.4**, se considerará una calidad de detección escasa.
- Si el índice Jaccard tiene un valor alrededor de **0.7**, se considerará una calidad de detección buena.
- Si el índice Jaccard tiene un valor alrededor de **0.9**, se considerará una calidad de detección excelente.

Figura 3.6: Evaluación loU

3.1.3. Entrenamiento

La principal diferencia entre el entrenamiento usando la técnica SSD y el entrenamiento de un detector típico, es que la información a cerca de loa cuadros delimitadores reales debe asignarse a salidas específicas dentro del conjunto fijo de salidas del detector. Tras realizar esta asignación, la función de pérdida y la propagación hacia atrás son aplicadas de extremo a extremo. El proceso de entrenamiento también implica elegir el conjunto de cuadros y escalas predeterminados para la detección.

- **Negativos:** Durante el entrenamiento, muchas de las *bounding boxes* tienen un nivel muy bajo de IoU y por lo tanto son interpretados como ejemplos de entrenamiento negativos. Esto provoca un significante desequilibrio entre ejemplos de entrenamiento positivos y negativos. En lugar de utilizar todos los ejemplos negativos, se ordenan usando la mayor pérdida de confianza para cada cuadro predeterminado y se seleccionan los más altos para asegurar que la relación de aspecto entre los negativos y los positivos sea como máximo 3:1. Esto provoca una optimización más rápida y un entrenamiento más estable.

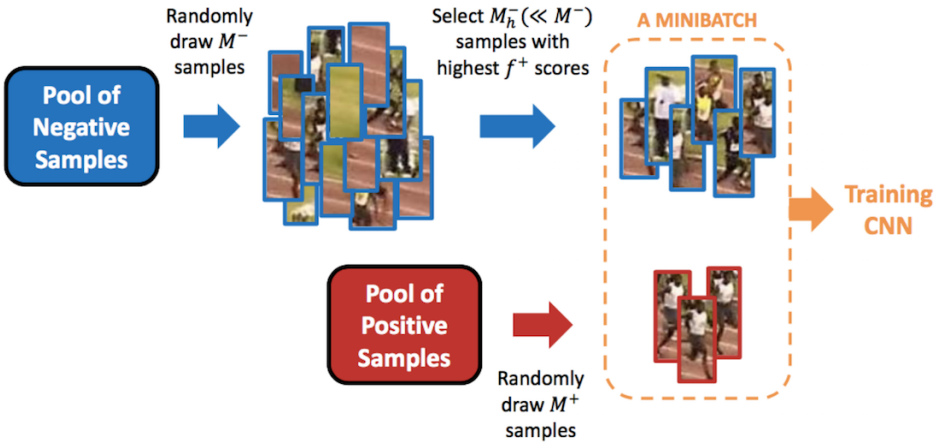

Figura 3.7: Ejemplo de uso de negativos en el entrenamiento SSD

- **Aumento de datos:** El aumento de datos, tanto en la técnica SSD como en muchas otras aplicaciones de aprendizaje profundo, ha sido crucial para que la red neuronal sea mucho más robusta frente a diferentes tamaños y formas de objetos a la entrada de ésta. Con este fin, se generaron ejemplos de entrenamiento con parches de la imagen original con diferentes relaciones de IoU (p.ej. 0.1, 0.3, 0.5, etc). Además, cada imagen se gira horizontalmente de forma aleatoria con una probabilidad de 0.5, asegurando así que los objetos aparezcan con la misma probabilidad a la izquierda que a la derecha.

3.2. Detector de objetos

3.2.1. Estructura de la red

En esta sección se explicará la estructura de la red neuronal, la cual entrenaremos con las bases de datos definidas en los apartados 2.4.1 y 2.4.2 pare realizar posteriormente la detección de objetos sobre una imagen de entrada.

En primer lugar, se especificará el nombre de la red, en este caso 'VGG_VOC0712_SSD_300x300_deploy'.

```
name: "VGG_VOC0712_SSD_300x300_deploy"
```

Al inicio, también es necesario definir la dimensiones que tendrán las imágenes usadas para el entrenamiento. Tanto en PascalVOC como COCO, las imágenes tienen una dimensión de 300x300 y un formato RGB, definiéndose en la red de la siguiente manera:

```
input_shape {
  dim: 1
  dim: 3
  dim: 300
  dim: 300
}
```

Como se ha explicado en la sección 3.2, la arquitectura de la red neuronal en la técnica SSD se basa en las redes convolucionales, por lo que a continuación se pueden observar varias capas de convolución combinadas con capas de pooling.

En primer lugar, se define la capa de convolución, cuyo funcionamiento se ha explicado en la sección 2.1.3.

```
layer {
  name: "conv1_1"
  type: "Convolution"
  bottom: "data"
  top: "conv1_1"
  param {
    lr_mult: 1.0
    decay_mult: 1.0
  }
  param {
    lr_mult: 2.0
    decay_mult: 0.0
  }
  convolution_param {
    num_output: 64
    pad: 1
    kernel_size: 3
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
```

```
      value: 0.0
    }
  }
}
```

Al inicio de esta capa se definen varios parámetros tales como el nombre de esta, *name*, el tipo, *type*, la capa de la cual recibe los datos, *bottom* y la capa a la cual enviará los datos procesados, *top*. Este ejemplo particular de capa de convolución generará 64 salidas, *num_output* y utilizará un núcleo de convolución de 3x3, *kernel_size*. Para inicializar los pesos se utilizará el algoritmo Xavier, el cual determina automáticamente la escala de inicialización basándose en el número de entradas y salidas de cada neurona. Adicionalmente, el sesgo se inicializará como una constante, siendo esta 0.0.

Posteriormente se define una capa de pooling, cuyo funcionamiento se ha explicado en el apartado 2.1.3.

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1_2"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

Esta capa tomará los datos de la capa de convolución anterior y utilizará bloques de 2x2 para dividir la imagen entrante. Para que no haya solapamiento entre bloques contiguos utilizará el parámetro *stride = 2*. Finalmente, como función de agrupación utilizará el máximo, escogiendo el píxel con mayor nivel de intensidad de cada bloque.

Entre las capas da convolución y pooling, se intercalan varias capas de activación, utilizando la función ReLU, explicada en el apartado 2.1.3.

```
layer {
  name: "relu1_1"
  type: "ReLU"
  bottom: "conv1_1"
```

```
  top: "conv1_1"
}
```

En esta capa simplemente se definirán los parámetros comunes que se definen en todas las capas de la red, es decir, nombre de ésta, tipo, capa de la que recibe los datos y capa a la cual se los enviará.

Además de las capas de activación, existirán varias capas de diferentes tipos que procesarán los datos obtenidos de diferentes maneras.

- Las capas de datos de tipo *Flatten* transforma una entrada de dimensiones *n * c * h * w* en un vector simple de dimensiones *n * (c*h*w)*

```
layer {
  name: "conv4_3_norm_mbox_loc_flat"
  type: "Flatten"
  bottom: "conv4_3_norm_mbox_loc_perm"
  top: "conv4_3_norm_mbox_loc_flat"
  flatten_param {
    axis: 1
  }
}
```

- Las capas de datos de tipo *Reshape* recibe como entrada una capa de unas dimensiones arbitrarias y genera como salida las misma capa pero con las dimensiones definidas en *reshape_param*.

```
layer {
  name: "mbox_conf_reshape"
  type: "Reshape"
  bottom: "mbox_conf"
  top: "mbox_conf_reshape"
  reshape_param {
    shape {
      dim: 0
      dim: -1
      dim: 21
    }
  }
}
```

Las dimensiones de salida están especificadas en *reshape_param*. Los números positivos se utilizan directamente, configurando la dimensión correspondiente en la capa de salida. Además, se aceptan 2 valores especiales para definir cualquier que se desee.

- Si se utiliza el valor **0** se copiará la dimensión de la capa inferior, es decir, si la capa inferior tiene 2 como su primera dimensión, la capa superior tendrá 2 como su primera dimensión.
- Si se utiliza **-1** se reducirá este valor de las otras dimensiones.

- Las capas de datos de tipo *Concat* concatena varias capas de entrada en una única capa de salida.

```
layer {
  name: "mbox_loc"
  type: "Concat"
  bottom: "conv4_3_norm_mbox_loc_flat"
  bottom: "fc7_mbox_loc_flat"
  bottom: "conv6_2_mbox_loc_flat"
  bottom: "conv7_2_mbox_loc_flat"
  bottom: "conv8_2_mbox_loc_flat"
  bottom: "conv9_2_mbox_loc_flat"
  top: "mbox_loc"
  concat_param {
    axis: 1
  }
}
```

Suponiendo que se recibe una entrada *n_i * c_i * h * w* para cada capa de entrada *it*.

- Si *axis = 0*, *(n_1 + n_2 + ... + n_K) * c_1 * h * w* y todas las entrada *c_i* deberían ser iguales.
- Si *axis = 1*, *(n_1 * (c_1 + c_2 + ... + c_K) * h * w* y todas las entrada *n_i* deberían ser iguales.

Finalmente, se define la capa final.

```
layer {
  name: "detection_out"
  type: "DetectionOutput"
```

```
  bottom: "mbox_loc"
  bottom: "mbox_conf_flatten"
  bottom: "mbox_priorbox"
  top: "detection_out"
  include {
    phase: TEST
  }
  detection_output_param {
    num_classes: 21
    share_location: true
    background_label_id: 0
    nms_param {
      nms_threshold: 0.449999988079
      top_k: 400
    }
    save_output_param {
      output_directory: "/home/davidbutra/data/VOCdevkit/
                  results/VOC2007/SSD_300x300/Main"
      output_name_prefix: "comp4_det_test_"
      output_format: "VOC"
      label_map_file: "data/VOC0712/labelmap_voc.prototxt"
      name_size_file: "data/VOC0712/test_name_size.txt"
      num_test_image: 4952
    }
    code_type: CENTER_SIZE
    keep_top_k: 200
    confidence_threshold: 0.00999999977648
  }
}
```

3.2.2. Definición del solucionador

```
train_net: "models/VGGNet/VOC0712/SSD_300x300/train.prototxt"
test_net: "models/VGGNet/VOC0712/SSD_300x300/test.prototxt"
test_iter: 619
test_interval: 10000
base_lr: 0.0010000000475
display: 10
max_iter: 120000
lr_policy: "multistep"
gamma: 0.10000000149
momentum: 0.899999976158
```

```
weight_decay: 0.000500000023749
snapshot: 80000
snapshot_prefix: "models/VGGNet/VOC0712/SSD_300x300
              /VGG_VOC0712_SSD_300x300"
solver_mode: CPU
device_id: 0
debug_info: false
snapshot_after_train: true
test_initialization: false
average_loss: 10
stepvalue: 80000
stepvalue: 100000
stepvalue: 120000
iter_size: 1
type: "SGD"
eval_type: "detection"
ap_version: "11point"
```

3.2.3. Entrenamiento de la red

3.3. SSD en Caffe

En primer lugar, se definen 3 ficheros que tendrán gran importancia a lo largo del proceso de detección:

- **labelmap.prototxt:** Definirá las etiquetas que pueden ser asignadas en la detección.
- **model_def:** Definirá la estructura básica de la red neuronal.
- **model_weights:** Se trata de un fichero binario que contiene el estado actual de los pesos para cada capa de la red neuronal.

Posteriormente, se define la función *get_labelname*, que es la encargada de devolver las etiquetas que se encuentran definidas en el fichero labelmap.prototxt.

```
def get_labelname(self,labelmap, labels):
    num_labels = len(labelmap.item)
    labelnames = []
    if type(labels) is not list:
        labels = [labels]
```

```
    for label in labels:
        found = False
        for i in xrange(0, num_labels):
            if label == labelmap.item[i].label:
                found = True
                labelnames.append(labelmap.item[i].display_name)
                break
        assert found == True
    return labelnames
```

Finalmente, se define la función *detection_test*, la cual toma una imagen como entrada y retorna esta misma imagen con las bounding boxes de los objetos detectados superpuestas.

- Primero, esta función realiza una serie de transformaciones para adaptar la imagen de entrada a la red neuronal entrenada. Para ello, se utiliza la función *Transformer* proporcionada por caffe.
- Después, la imagen transformada se introduce en la red usando el siguiente código.

```
self.net.blobs['data'].data[...] = transformed_image
```

- Después de esto, comienza la parte de la detección. Para ello, se utiliza la siguiente función de Caffe, la cual devuelve las detecciones obtenidas.

```
detections = self.net.forward()['detection_out']
```

- Esta salida es parseada en varios vectores que contienen las etiquetas de los objetos detectados, su confianza y las coordenadas de su *bounding box*.

```
det_label = detections[0,0,:,1]
det_conf = detections[0,0,:,2]
det_xmin = detections[0,0,:,3]
det_ymin = detections[0,0,:,4]
det_xmax = detections[0,0,:,5]
det_ymax = detections[0,0,:,6]
```

- Es importante no seleccionar todas las detecciones obtenidas, ya que algunas de ellas con un valor de confianza bajo pueden no ser tan precisas como se desee. Para ello, se usa un filtro que elige las detecciones con un nivel de confianza mayor a 0.6.

```
# Get detections with confidence higher than 0.6.
for i in range(0, len(det_conf)):
    if (det_conf[i] >= 0.6):
        top_indices.append(i)

top_conf = det_conf[top_indices]
top_label_indices = det_label[top_indices].tolist()
top_labels = self.get_labelname(self.labelmap,
                      top_label_indices)
top_xmin = det_xmin[top_indices]
top_ymin = det_ymin[top_indices]
top_xmax = det_xmax[top_indices]
top_ymax = det_ymax[top_indices]
```

3.4. Comparación *bounding boxes* real y detectada

En el apartado 3.3 se ha explicado como se obtiene la *bounding box* de los objetos detectados. Para obtener la *bounding box* real, hay que recuperar las coordenadas del objeto, las cuales se encuentran en las anotaciones de la imagen. Como se ha explicado en el apartado 2.4, las anotaciones para PascalVOC y COCO tienen formatos de archivos diferentes, para la primera base de datos se trata de ficheros XML y para la segunda ficheros JSON, por lo que se utilizarán 2 scripts diferentes para recorrer el fichero determinado y obtener las coordenadas originales. Tras ello, se podrá observar fácil y gráficamente la diferencia entre la *bounding box* real y la detectada, consiguiendo así una aproximación visual de la calidad del detector.

3.4.1. PascalVOC

Se define la función *get_ground_truth*, la cual devolverá un array por cada objeto detectado en el que vendrán informados las coordenadas de la *bounding box* real, y la etiqueta del objeto determinado. Esta función recibirá como datos de entrada la ruta de la imagen de entrada, un array con las

etiquetas de los objetos detectados y las coordenadas de las *bounding boxes* de éstos.

En primer lugar, se utilizará la ruta de la imagen de entrada recibida (p.ej '/home/db/data/VOCdevkit/VOC2012/JPEGImages/2007_009938.jpg') para obtener la ruta del fichero con las anotaciones correspondientes a esa imagen. Para ello, se utilizará el siguiente código:

```
annotation = input_image.split(".")

annotation = annotation[0].split("/")

file_annotations = "/" + annotation[1] + "/" + annotation[2]
+ "/" + annotation[3] + "/" + annotation[4] + "/" + annotation[5]
+"/" + 'Annotations' + "/" + annotation[7] + ".xml"
```

Realizando los 2 *split* tendremos todas las partes de la ruta separadas dentro de la variable *annotation*. Posteriormente, concatenando cada parte de la ruta con /", sustituyendo *JPEGImages* por *Annotations*, e incluyendo el formato de los ficheros de anotaciones, es decir, *xml*, tendremos la ruta del fichero de anotaciones correspondiente a la imagen de entrada.

Posteriormente, se utilizarán comandos *Python* creados para la lectura de ficheros *XML*, específicamente *parse* y *getroot*.

```
tree = ET.parse(file)
root = tree.getroot()
```

A partir de este código, tendremos acceso a los elementos del fichero de anotaciones utilizando para ello la variable *root*.

Después de esto, comenzará el primer bucle de esta función. Éste será el encargado de guardar en una variable varios datos de cada uno de los objetos existentes en el fichero de anotaciones. En primer lugar, el bucle buscará y recorrerá todas las etiquetas *object* del *xml*. Para ello se utilizará el comando de Python *findall* sobre la variable *root* definida anteriormente.

```
for element in root.findall('object'):
```

Luego, con el comando *find*, accederemos a las etiquetas *name* y *bbox* del fichero de anotaciones, guardando su valor en el array *names_xml* definido

previamente.

```
name = element.find('name').text
names_xml.append(name)
bndbox = element.find('bndbox')
names_xml.append(bndbox[0].text) #xmin
names_xml.append(bndbox[1].text) #ymin
names_xml.append(bndbox[2].text) #xmax
names_xml.append(bndbox[3].text) #ymax
```

Para mejorar el acceso posterior a estos datos, éstos se guardarán en un array de arrays, en el que cada array interno tendrá los datos mencionados anteriormente. Para ello, se definirá el array *names_xml_final* y se guardará el array *names_xml* dentro de éste. Posteriormente, se eliminarán los datos guardados en el array *names_xml*.

```
names_xml_final.append(names_xml)
names_xml = []
```

Tras finalizar este bucle, comenzará el código en el que se comparan las *bounding boxes* reales con las detectadas para averiguar cuál es el cuadro delimitador real correspondiente para cada uno de los objetos detectados. En primer lugar se definirá un bucle, que recorrerá el array con todas las etiquetas de los objetos detectados el cual se pasó como parámetro a la función.

```
for x in range(len(label_detection)):
```

Después, se definirán 2 bucles más. El primero recorrerá el array con las coordenadas de los objetos detectados pasada a la función como parámetros. El segundo recorrerá el array *names_xml_final* definido anteriormente.

```
for t in range(len(ground_detection)):
    for i in range(len(names_xml_final)):
```

Ahora, el código comparará las coordenadas reales con las detectadas. Específicamente, se verificará que cada coordenada *xmin*, *ymin*, *ymax*, *ymax* de cada *bounding box* de los objetos detectados no exceda en 40 píxeles con las coordenadas originales.

```
if abs(int(ground_detection[t][0])-int(names_xml_final[i][1]))<=40
and abs(int(ground_detection[t][1])-int(names_xml_final[i][2]))<=40
and abs(int(ground_detection[t][2])-int(names_xml_final[i][3]))<=40
```

```
and abs(int(ground_detection[t][3])-int(names_xml_final[i][4]))<=40:
```

Con esto se conseguirá que cada *bounding box* detectada se asigne correctamente a la *bounding box* real, ya que si en el fichero de anotaciones existen varios objetos con la misma etiquetas, se podrían realizar asignaciones incorrectas si solo se compara la etiqueta y se obvian las coordenadas.

Posteriormente, tras verificar que las coordenadas son las correctas, se realizará la última comprobación. Ésta estará relacionada con la etiquetas de los objetos reales, almacenadas en la variable *names_xml_final*, y las etiquetas de los objetos detectados, almacenadas en la variable *label_detection*, la cual fue pasada como parámetro a la función.

```
if names_xml_final[i][0] == label_detection[x]
```

Tras esto, si se cumplen las condiciones, se asignarán a diferentes variables las coordenadas existentes en el fichero de anotaciones, es decir, las reales.

```
xmin = names_xml_final[i][1]
ymin = names_xml_final[i][2]
xmax = names_xml_final[i][3]
ymax = names_xml_final[i][4]
```

Finalmente, la función devolverá el array *positions_array*, el cuán contendrá todas las variables comentadas anteriormente.

```
positions = []
positions.append(xmin)
positions.append(ymin)
positions.append(xmax)
positions.append(ymax)
positions_array.append(positions)
```

Finalmente, las *bounding boxes* reales se pintarán sobre la imagen de entrada a partir del array devuelto en la función anterior utilizando el siguiente código:

```
for i in range(len(ground_truth)):
    xmin = int(round(float(ground_truth[i][0])))
    ymin = int(round(float(ground_truth[i][1])))
    xmax = int(round(float(ground_truth[i][2])))
```

```
    ymax = int(round(float(ground_truth[i][3])))

    cv2.rectangle(img,(xmin,ymin),(xmax,ymax),(0,255,0),2)#Color BGR
```

La imagen 3.8 muestra un ejemplo de una salida que generaría este *script*. La *bounding box* de color verde hace referencia a las coordenadas reales del objeto, mientras que la *bounding box* de color rojo encuadra el objeto detectado.

Figura 3.8: Comparacion bounding boxes real y detectada con imagen de la BBDD PascalVOC

3.4.2. COCO

Como se ha explicado en la sección 2.4.2, las anotaciones de esta base de datos están almacenadas en archivos de formato JSON, por lo que para conseguir las coordenadas reales de los objetos habrá que utilizar comandos que permitan recorrer este tipo de ficheros. Antes de realizar la función principal que devuelva la imagen con las *bounding boxes* real y detectada superpuestas, se crearán varios métodos que recorrerán el fichero de anotaciones, preparando así los datos para ser tratados posteriormente.

En primer lugar, se definirá la función *get_image_id*, la cual se encargará de obtener el *id* de la imagen de entrada, que será necesario para encontrar las anotaciones correspondientes a ésta. Este *id* se puede conseguir a partir del *path* de la imagen *jpg*, el cual se pasa como parámetro de la función.

```
def get_image_id(self, image_file):

        image_file = re.split('.jpg', image_file)
        image_file = re.split('_', image_file[0])
        image_id = int(image_file[2])

        return image_id
```

La segunda función para la preparación de los datos se llamará *annotations* y se encargará se recuperar todas las anotaciones de la imagen de entrada. Para ello se le pasará como parámetro el *id* de la imagen calculado en la función explicada anteriormente.

Primero, se guardará en la variable *jsonFileAnnotations* los objetos con nombre *annotations* de todo el fichero de anotaciones *jsonFile*, definido como variable global del script.

```
jsonFileAnnotations = jsonFile["annotations"]
```

En la imagen 2.7, se puede observar todos los pares de nombre/valor que tiene el objeto *annotations*. En este script, se utilizará el parámetro con nombre *image_id*, comparando uno a uno el existente en cada uno de los objetos *annotations* de la variable *jsonFile* con el *id* de la imagen de entrada, el cual se pasa a esta función como parámetro.

```
if c["image_id"] == image_id:
    if len(c['bbox']) == 1:
```

Para realizar posteriormente la comparación con los objetos detectados necesitaremos las coordenadas y etiquetas reales de todos los objetos existentes en la imagen. Estos datos se encuentran en los parámetros *bbox* y *category_id* del fichero de anotaciones los cuales se guardarán en el array *annotations* definido anteriormente. Para que la función devuelva un array de arrays con todas las coordenadas y etiquetas, añadiremos el array *annotations* en otro array definido llamado *annotations_final*.

```
annotations.append(c['category_id'])
annotations.append(c['bbox'][0][0])
annotations.append(c['bbox'][0][1])
annotations.append(c['bbox'][0][2])
annotations.append(c['bbox'][0][3])
annotations_final.append(annotations)
annotations = []
```

El objetivo de la tercera función será recuperar todas las etiquetas e *ids* de la base de datos. Para ello, como anteriormente, se buscará un objeto en todo el fichero JSON de anotaciones, en este caso, el llamado *categories*.

```
jsonFileCategories = jsonFile["categories"]
```

Posteriormente, la función recorrerá la variable *jsonFileCategories* recuperando los parámetros del fichero llamados *name* y *id*, los cuales hacen referencia a las etiquetas y a su *id*.

```
for c in jsonFileCategories:
    labels_ids_array.extend([c["name"], c["id"]])
        labels_ids_array_total.append(labels_ids_array)
        labels_ids_array = []
```

La cuarta y última función, se trata de *recover_id* y se encargará de recuperar el *id* de los objetos detectados. Este *id* es un número entero que hace referencia a cada uno de los objetos existentes en la base de datos COCO. A esta función se le pasarán como parámetros 2 arrays. El primero de ellos hará referencia a todas las etiquetas de cada objeto detectado anteriormente. El segundo se referirá a todos los *ids* y nombre de etiquetas de la base de datos recuperados anteriormente.

```
    def recover_id(self, labels_detected_array, labels_ids_array):
        id_array = []
        for c in range(len(labels_detected_array)):
            for i in range(len(labels_ids_array)):
                print labels_ids_array[i]
                if labels_detected_array[c] == labels_ids_array[i][0] :
                    id_array.append(labels_ids_array[i][1])
                    break

        return id_array
```

Esta función recorrerá ambos arrays comparando sus registros. En el momento en que coincidan los nombres de etiquetas, se le asignará a ese objeto detectado el *id* determinado.

Tras estas 4 funciones, finalmente se llamará al método *get_ground_truth* la cual devolverá un array con las coordenadas reales de los objetos detectados. Este función recibirá como entrada las variables *id_array*, *ground_detection*, *image_id* y *image_annotations* las cuales son los resultados de las 4 funciones explicadas anteriormente.

```
def get_ground_truth(self, id_array,
                ground_detection,
                image_id,
                image_annotations):
```

Esta función recorrerá 2 arrays. Primero se definirá el bucle que recorre el array *id_array* que contiene los *ids* y etiquetas de los objetos detectados. Después se definirá el bucle que recorre el array *image_annotations* que contiene las anotaciones referentes a la imagen de entrada. Dentro de estos 2 bucles su realizarán las comprobaciones necesarias para asignar a cada *bounding box* detectada su *bounding box* real.

```
for i in range(len(id_array)):
    for c in range(len(image_annotations)):
```

Se compararán los *ids* detectados con los *ids* existentes en las anotaciones. En el momento en el qué coincidan los *ids*, se asignaran a las variables *xmin*, *ymin*, *xmax* y *ymax* las coordenadas de la *bounding box real*.

```
xmin = int(image_annotations[c][1])
ymin = int(image_annotations[c][2])
xmax = xmin + int(image_annotations[c][3])
ymax = ymin + int(image_annotations[c][4])
```

En la base de datos COCO, a diferencia de PascalVOC, los 2 últimos valores referentes a las coordenadas, definen la anchura y altura de las *bounding boxes* por lo que para conseguir los valores correctos de *xmax* e *ymax* habrá que sumar a *xmin* e *ymin* la anchura y altura respectivamente.

La asociación de *ids* realizada anteriormente no garantiza que ese objeto definido en las anotaciones haga referencia al objeto detectado. En la imagen

de entrada y en las anotaciones pueden existir varias instancias de objetos con el mismo identificador, por lo que es necesario realizar otra comprobación adicional.

```
if abs(xmin - ground_detection[i][0]) <= 90 and
abs(ymin - ground_detection[i][1]) <= 90 and
abs(xmax - ground_detection[i][2]) <= 90 and
abs(ymax - ground_detection[i][3]) <= 90:
```

De esta manera, se comprueba que la diferencia entre las coordenadas reales y detectadas del objeto no es mayor de 90 píxeles, asegurando de esta manera que la *bounding box* real y la *bounding box* detectada se refieren al mismo objeto de la imagen.

Tras realizar esta comprobación, se añade en el array *ground_truth_final*, el cual será lo que devuelva la función explicada, el array *ground_truth* formado por las coordenada reales.

```
ground_truth.append(xmin)
ground_truth.append(ymin)
ground_truth.append(xmax)
ground_truth.append(ymax)
ground_truth_final.append(ground_truth)
ground_truth = []
```

Finalmente, las *bounding boxes* reales se pintarán sobre la imagen de entrada a partir del array devuelto en la función anterior utilizando el siguiente código:

```
for i in range(len(ground_truth)):
    xmin = ground_truth[i][0]
    ymin = ground_truth[i][1]
    xmax = ground_truth[i][2]
    ymax = ground_truth[i][3]

    cv2.rectangle(img,(xmin,ymin),(xmax,ymax),(0,255,0),2)#Color BGR
```

La imagen 3.9 muestra un ejemplo de una salida que generaría este *script*. La *bounding box* de color verde hace referencia a las coordenadas reales del objeto, mientras que la *bounding box* de color rojo encuadra el objeto detectado.

Figura 3.9: Comparación bounding boxes real y detectada con imagen de la BBDD COCO

3.5. Medidor de calidad

3.5.1. Calculo del índice Jaccard

Se partirá de los *scripts* explicados en los apartados 3.4.1 y 3.4.2 pero ahora los scripts crearán un fichero mostrando el índice Jaccard calculado para cada uno de los objetos de las 2 bases de datos. Estos 2 nuevos *scripts*, a diferencia de los anteriores, recibirán un conjunto de imágenes, por lo que el primer paso para realizarlos será preparar los datos de entrada.

- Para el script que evaluará las imágenes de la base de datos PascalVOC, utilizaremos la ruta a las imágenes y a los ficheros de anotaciones. A partir de esto, se crearán 2 variables las cuales contendrán un listado con todos los archivos.

```
dirsImages = sorted(os.listdir(pathImages))
dirsAnnotations = sorted(os.listdir(pathAnnotations))
```

Posteriormente, se iniciará un bucle en el que se llamará a la función *detectiontest* tantas veces como se desee.

```
for x in range(len(dirsImages)):
    if x == 200:
        break
    image = cv2.imread(pathImages + dirsImages[x])
    img_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    img_rgb = cv2.cvtColor(img_gray,cv2.COLOR_GRAY2RGB)
    detec = net.detectiontest(image,
                      pathAnnotations + dirsAnnotations[x])
    detec_array.append(detec)
```

A esta función se le pasará como parámetro la imagen de entrada y su ruta y devolverá el índice Jaccard obtenido para cada uno de los objetos de la imagen de entrada. Después, introducirá estos valores en la variable *detec_array*.

- Para el script que evaluará las imágenes de la base de datos COCO, la única diferencia será que solo habrá que listar la carpeta donde se encuentran las imágenes. En esta base de datos existe un único fichero de anotaciones que contiene todos los datos reales de cada una de las imágenes de ella por lo que no será necesario asociar cada imagen a un fichero de anotaciones. Posteriormente, se llamará al bucle de la misma manera que se ha comentado en el punto anterior.

Con los *scripts* explicados en los apartados 3.4.1 y 3.4.2 tenemos disponibles las *bounding boxes* reales y detectadas necesarias para calcular el índice Jaccard. Ahora, los 2 nuevos *scripts* creados no pintarán estas cajas delimitadoras dentro de la imagen si no que las utilizarán para calcular el índice. Para ello, se crearán 4 nuevas funciones, las cuales se explican a continuación.

La primera de ellas se llamará *area*, la cual recibirá un array con las coordenadas de los rectángulos sobre los cuales se quiere conocer su área y devolverá un array con los valores obtenidos.

```
def area(self, array_positions):
    area_array = []
    for i in range(len(array_positions)):
        weight = float(array_positions[i][2]) -
```

```
                float(array_positions[i][0])
            height = float(array_positions[i][3]) -
                float(array_positions[i][1])
            area = weight * height

            area_array.append(area)

        return area_array
```

A esta función se la llamará 2 veces, guardando los valores obtenidos en las variables *truth_area* y *detection_area* las cuales se referirán al area de las *bounding boxes* reales y detectadas respectivamente.

El segundo método se llamará *rectangle_intersection* y se encargará de calcular las coordenadas del rectángulo que forman las *bounding boxes* real y detectada cuando intersectan. Para ello, se le pasará como parámetro las coordenadas reales y detectadas para cada uno de los objetos localizados en la imagen. En esta función se realizarán varias comprobaciones con el objetivo calcular las coordenadas del rectángulo intersección.

- La coordenada *xmin* del rectángulo intersección será la coordenada *xmin* **mayor** de los 2 rectángulos que intersectan.

```
if rectangles_array1[0][x][0]) >= rectangles_array2[x][0]:
    xmin_intersection = rectangles_array1[0][x][0]
else:
    xmin_intersection = rectangles_array2[x][0]
```

- La coordenada *ymin* del rectángulo intersección será la coordenada *ymin* **mayor** de los 2 rectángulos que intersectan.

```
if rectangles_array1[0][x][1] >= rectangles_array2[x][1]:
    ymin_intersection = rectangles_array1[0][x][1]
else:
    ymin_intersection = intrectangles_array2[x][1]
```

- La coordenada *xmax* del rectángulo intersección será la coordenada *xmax* **menor** de los 2 rectángulos que intersectan.

```
if rectangles_array1[0][x][2] <= rectangles_array2[x][2]:
    xmax_intersection = rectangles_array1[0][x][2]
else:
    xmax_intersection = rectangles_array2[x][2]
```

- La coordenada *ymax* del rectángulo intersección será la coordenada *ymax* **menor** de los 2 rectángulos que intersectan.

```
if rectangles_array1[0][x][3] <= rectangles_array2[x][3]:
    ymax_intersection = rectangles_array1[0][x][3]
else:
    ymax_intersection = rectangles_array2[x][3]
```

Finalmente, la función devolverá el array *positions_array* el cual contendrá un array con las coordenadas de cada uno de los rectángulos calculados anteriormente.

```
positions = []
positions.append(xmin_intersection)
positions.append(ymin_intersection)
positions.append(xmax_intersection)
positions.append(ymax_intersection)
positions_array.append(positions)
```

Después de esto, se llamará a la función *area*, explicada anteriormente, para obtener el área de los rectángulos intersección, creando así la variable *area_intersection*.

El último parámetro necesario para calcular el índice Jaccard, es el área total de la *bounding box* real y detectado. Para ello, se utiliza el método *area_total*, el cual recibirá como parámetro las coordenadas reales y detectadas de cada *bounding box*, y el área de los rectángulos intersección calculados anteriormente y almacenados en la variable *area_intersection*.

```
def area_total(self, rectangle1, rectangle2, area_intersection):
```

Esta función recorrerá todos lo rectángulos asociados, calculando en primer lugar el área de cada uno de ellos.

```
heigh_rectangle1 = rectangle1[0][x][3] - rectangle1[0][x][1]
weigh_rectangle1 = rectangle1[0][x][2] - rectangle1[0][x][0]
area_rectangle1 = heigh_rectangle1 * weigh_rectangle1

heigh_rectangle2 = rectangle2[x][3] - rectangle2[x][1]
weigh_rectangle2 = rectangle2[x][2]) - int(rectangle2[x][0]
area_rectangle2 = heigh_rectangle2 * weigh_rectangle2
```

El área total se calculará sumando el área total de los 2 rectángulos y eliminando posteriormente el área del rectángulo intersección, el cual se ha pasado a la función como parámetro. Por último, se agregará al array *area_total_array* el valor obtenido.

```
area_total=area_rectangle1+area_rectangle2-area_intersection[x]

area_total_array.append(area_total)
```

Tras realizar estos métodos, se dispondrán de todos los datos necesarios para el cálculo del índice Jaccard. Como se explica en el apartado 3.1.2 y se muestra gráficamente el la figura 3.5, el índice Jaccard se calcula dividiendo la intersección de los 2 conjuntos entre la unión de ambos, es decir, el área del rectángulo intersección entre el área total. Para ello, se utilizará la función *jaccard_index* la cual recibirá como parámetro las variables *area_intersection* y *area_total*, que contienen los datos necesarios para el cálculo del índice.

```
def jaccard_index(self, area_total, area_intersection):
```

Este método recorrerá un bucle cuya longitud será el total del conjunto de *bounding boxes* reales y detectadas para cada objeto identificado, realizando la división comentada anteriormente. Finalmente, el resultado se devolverá en la variable *jaccard_index_array*.

```
jaccard_index = float(area_intersection[x]) / float(area_total[x])

jaccard_index_array.append(jaccard_index)
```

Una vez conseguido el valor del índice Jaccard, la función *detectiontest* devolverá 2 arrays. El primero con el valor del índice Jaccard obtenido y el segundo con todas las etiquetas de los objetos detectados. Estos 2 arrays se pueden relacionar con el índice, ya que el valor del índice Jaccard en una determinada posición del array es el obtenido para el objeto existente en la

misma posición del segundo array.

Como se ha comentado al principio de esta sección, ambos *scripts*, tanto el que utiliza las imágenes de entrada de PascalVOC como el que utiliza las de COCO, llamarán a la función *detectiontest* una vez por cada imagen de entrada, guardando el resultado obtenido en cada llamada al método en la variable *detec_array*.

Cuando se obtengan todos los valores del índice Jaccard que se desee, se generarán 2 ficheros, que mostrarán las estadísticas de los datos obtenidos. El primer fichero se llamará *jaccardIndex.txt* y mostrará el índice Jaccard para cada uno de los objetos detectados en cada una de las imágenes. Para ello se recorrerá la variable *detec_array* relacionando los índices de éste.

```
if detec_array[x][1][i] == "aeroplane":
    aeroplane_jaccard = aeroplane_jaccard + detec_array[x][0][i]
    naeroplane = naeroplane + 1
```

El código anterior comparará una variable creada con el nombre de una de las etiquetas de las base de datos con la etiqueta de cada objeto detectado,

3.5.2. Resultado final

3.6. Componente Python

Este componente se compondrá de 3 hilos de ejecución.

- Hilo encargado de capturar las imágenes de la red utilizando para ello el servidor de imágenes, *cameraserver*, explicado en el apartado 2.2.1. Lee de la red y las vuelca en una memoria A.
- Hilo encargado de realizar la detección. Para ello, coge una de las imágenes de entrada, almacenadas en memoria A, realiza la detección y escribe su resultado en memoria B.
- Hilo del GUI, en el cual se visualiza todo, tanto la imagen almacenada en memoria A como la almacenada en memoria B.

3.6.1. Camera

3.6.2. Detector

3.6.3. GUI

3.6.4. Ejecución

```
cameraserver cameraserver.cfg
```

```
python detectorSSD.py --Ice.Config=detectorSSD.cfg
```

Bibliografía

[1] Edgar Nelson Sánchez Camperos y Alma Yolanda Alanís García: Redes Neuronales: Conceptos fundamentales y aplicaciones a control automático"

[2] Wei Liu ,Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed , Cheng-Yang Fu, Alexander C. Berg: SSD: Single Shot MultiBox Detector