



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE
TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN SISTEMAS
AUDIOVISUALES Y MULTIMEDIA

TRABAJO FIN DE GRADO

Detección de Objetos en Imágenes con Caffe

Autor: David Butragueño Palomar

Tutor: José María Cañas Plaza

Cotutor: Inmaculada Mora Jiménez

Curso académico 2018/2019

Agradecimientos

Resumen

Durante este trabajo se ha abordado el tema de la Inteligencia Artificial, más en concreto del *Deep Learning*, explicando sus características y funcionalidades que hacen posible crear modelos o sistemas que simulen el comportamiento humano. El objetivo principal del trabajo es entrenar una red neuronal y la posterior medición y comparación de prestaciones de ésta con otros modelos preentrenados a la hora de realizar la detección de objetos.

En primer lugar, se realiza un estudio de la infraestructura que se ha utilizado durante la realización del proyecto. Se definen las principales características de las herramientas Caffe y JdeRobot, utilizadas para el entrenamiento del modelo y para la creación de un componente detector en tiempo real. Posteriormente, se realiza un estudio de las bases de datos que se usarán, PASCAL Visual Object Classes y Common Objects in Context, especificando el número total de imágenes, como están distribuidos los objetos y el formato de sus ficheros de anotaciones.

Tras la definición de la infraestructura, se realiza una introducción a la técnica *Single Shot MultiBox Detection*, la cual será la usada en la red neuronal que se entrenará. Despues, se muestra un ejemplo práctico en el que se integra Caffe en esta técnica. Tras esto, se explica todo el proceso de entrenamiento del modelo, explicando la estructura de la red y los parámetros con los que realizará dicho entrenamiento. Por último, se explica como se desarrolló el componente detector comentado anteriormente y se muestra un ejemplo de su ejecución.

Finalmente, se introduce el concepto de índice Jaccard, se procede a realizar las pruebas necesarias para comparar las prestaciones tanto de la red entrenada como de los modelos ya existentes comentados anteriormente y se exponen las conclusiones tras la realización del proyecto además de posibles líneas futuras a seguir.

Índice general

Índice de figuras	VII
Índice de tablas	IX
Acrónimos	XI
1 Introducción	1
1.1 Inteligencia artificial y aprendizaje automático	1
1.2 Redes neuronales artificiales	4
1.2.1 Aprendizaje en redes neuronales artificiales	7
1.2.2 Redes neuronales convolucionales (CNN)	8
1.2.3 Entornos software para redes neuronales	9
1.2.4 Deep Learning	11
1.3 Antecedentes en Robotics de la URJC	13
2 Objetivos y metodología	17
2.1 Objetivos	17
2.2 Metodología y plan de trabajo	18
3 Infraestructura utilizada	21
3.1 Plataforma Caffe	21
3.1.1 Interfaz vía comandos	22
3.1.2 Interfaz con lenguaje Python	22
3.1.3 Capas de una red neuronal	23
3.2 Organización JdeRobot	28
3.2.1 Servidor de imágenes CameraServer	29
3.2.2 DetectionSuite	30

3.3 Bases de datos de detección visual	31
3.3.1 PASCAL Visual Object Classes	31
3.3.2 Common Objects in Context	34
4 Detección visual de objetos con Single Shot MultiBox y Caffe	37
4.1 Técnica Single Shot MultiBox Detection	37
4.1.1 Arquitectura	39
4.1.2 Entrenamiento	40
4.2 Aplicación para detección de objetos en flujos de vídeo con SSD	43
4.2.1 Diseño	43
4.2.2 Hilo Camera	44
4.2.3 Hilo Detector	45
4.2.4 Hilo GUI	46
4.2.5 Ejecución	47
4.2.6 Detección específica de un tipo de objetos	48
4.3 Uso de redes SSD preentrenadas en Caffe	49
4.4 Creación y uso de un modelo de red SSD propio	53
4.4.1 Estructura de la red	53
4.4.2 Definición del solucionador	59
4.4.3 Entrenamiento de la red	61
5 Medidor de calidad	63
5.1 Comparación <i>bounding boxes</i> real y detectada	63
5.1.1 PascalVOC	63
5.1.2 COCO	68
5.2 Métricas de evaluación	72
5.2.1 Precisión y Recall	73
5.2.2 Índice Jaccard	73
5.3 Resultado final	75
6 Conclusiones	83
6.1 Conclusiones	83
6.2 Líneas futuras	85
Bibliografía	87

Índice de figuras

1.1	Modelo Machine Learning [5].	3
1.2	Ejemplo de AA aplicado a la detección de cáncer de mama [6].	4
1.3	Modelo de una neurona artificial [8].	5
1.4	Modelo de una red neurona artificial [10].	6
1.5	Modelo de una red neurona convolucional [11].	8
1.6	Ejemplo de campo receptivo local en la capa convolucional [11].	9
1.7	Marco de la Inteligencia Artificial [16].	12
1.8	Comparación de un enfoque de aprendizaje automático para la categorización de vehículos (izquierda) con el aprendizaje profundo (derecha) [17]. . .	13
1.9	(a) Muestras de la base de datos MNIST. (b) Muestras de imágenes aplicando el filtro Sobel.	14
1.10	Ejemplo del componente clasificador.	14
1.11	Ejemplo del componente clasificador.	15
1.12	Herramienta DetectionSuite.	16
3.1	Ejemplo convolución con tamaño del núcleo 2x2.	24
3.2	<i>Pooling</i> con función de agrupación del máximo.	25
3.3	Función de activación ReLU.	27
3.4	Función de activación Sigmoide.	28
3.5	Distribución de objetos en PascalVOC [26].	32
3.6	Estructura básica de anotaciones en COCO [28].	34
3.7	Estructura básica de anotaciones en COCO [28].	35
3.8	Distribución de objetos en COCO.	36
4.1	Framework SSD [21].	38
4.2	Arquitectura SSD [21].	39
4.3	Arquitectura VGG-16 [30].	40
4.4	Ejemplo de uso de negativos en el entrenamiento SSD [30].	42

ÍNDICE DE FIGURAS

4.5	Arquitectura de la aplicación de detección visual con SSD.	44
4.6	Ejecución de la aplicación usando el botón de detección única.	48
4.7	Ejecución del componente detector condicional usando el botón de detección única.	49
4.8	(a) Detección con modelo entrenado con PascalVOC. (b) Detección con modelo entrenado con COCO.	52
4.9	Ejecución del entrenamiento de la red neuronal.	62
5.1	Comparacion bounding boxes real y detectada con imagen de la BBDD PascalVOC.	67
5.2	Comparación bounding boxes real y detectada con imagen de la BBDD COCO.	72
5.3	Intersección y unión sobre dos conjuntos A y B.	74
5.4	Intersection over Union [33].	74
5.5	Evaluación IoU [33].	75
5.6	Distribución de las imágenes de test.	76
5.7	Interfaz gráfica de la herramienta <i>Detector</i>	77
5.8	Ejecución de la herramienta <i>Detector</i>	78
5.9	Interfaz gráfica de la herramienta <i>Evaluator</i>	79

Índice de tablas

5.1 Tiempo requerido para la evaluación de cada modelo.	80
5.2 mAP conjunto para 10 umbrales de IoU de los 3 modelos evaluados.	80
5.3 mAR conjunto para 10 umbrales de IoU de los 3 modelos evaluados.	81
5.4 Comparativa conjunta de mAP y mAR de la media de los 10 umbrales de IoU para cada uno de los 3 modelos evaluados.	81
5.5 mAP medio de los 10 umbrales de IoU para cada uno de los objetos en las 3 redes evaluadas.	82
5.6 mAR medio de los 10 umbrales de IoU para cada uno de los objetos en las 3 redes evaluadas.	82

Acrónimos

AA Aprendizaje Automático.

AP Aprendizaje Profundo.

API Application Programming Interface.

CNN Convolutional Neural Network.

COCO Common Objects in Context.

GUI Graphical User Interface.

IA Inteligencia Artificial.

IoU Intersection over Union.

JSON JavaScript Object Notation.

LMDB Lightning Memory-Mapped Database.

mAP mean Average Precision.

mAR mean Average Recall.

PascalVOC PASCAL Visual Object Classes.

ReLU Rectified Linear Unit.

RNA Red Neuronal Artificial.

SSD Single Shot MultiBox Detector.

XML eXtensible Markup Language.

Capítulo 1

Introducción

Este Trabajo Fin de Grado se encuadra en la detección de objetos dentro de imágenes usando redes neuronales. En este capítulo se presentará el contexto de las redes neuronales, que son una rama de la inteligencia artificial que está ofreciendo muy buenos resultados en los últimos años.

1.1. Inteligencia artificial y aprendizaje automático

El deseo de crear y desarrollar aplicaciones que simulen el comportamiento humano es algo que lleva existiendo desde hace mucho tiempo. Durante las últimas décadas, ha ido aumentando gradualmente el progreso en la construcción de máquinas inteligentes que puedan ejecutar ciertas tareas tal y como lo haría un ser humano. Este área se denomina Inteligencia Artificial IA [1].

Para llevar a cabo este propósito, dentro del ámbito de la IA, han surgido varias técnicas como el *Machine Learning* o aprendizaje automático AA y más recientemente el *Deep Learning* o aprendizaje profundo (AP) el cual surgió para solucionar las limitaciones que presentaba el AA. El objetivo de estas técnicas es crear algoritmos con los que las máquinas sean capaces de *aprender* ciertos comportamientos.

Este trabajo se encuadra en el contexto del AP, con el objetivo de detectar objetos en imágenes. Durante el desarrollo del mismo, y debido a la utilización del *Deep Learning*, se utilizarán las redes neuronales artificiales (RNA), más concretamente el tipo llamado redes

1.1. INTELIGENCIA ARTIFICIAL Y APRENDIZAJE AUTOMÁTICO

neuronales convolucionales (CNN, del inglés *Convolutional Neural Network*), y la técnica *Single Shot MultiBox Detection* (SSD), conceptos que serán explicados y desarrollados más adelante.

El concepto de IA no tiene una definición específica. No existe un consenso entre los científicos e ingenieros que tratan el problema de la IA. Tal vez, una de las definiciones que se puede considerar más ajustada es la reflejada en la *Encyclopedia of Artificial Intelligence* [2]:

La IA es un campo de la ciencia y la ingeniería que se ocupa de la compresión, desde el punto de vista informático, de lo que se denomina comúnmente comportamiento inteligente. También se ocupa de la creación de artefactos que exhiben ese comportamiento.

En el seno de la IA como ciencia y tecnología se han ido acumulando conocimientos sobre cómo emular las diversas capacidades del ser humano para exhibir comportamientos inteligentes y se han desarrollado sistemas cada vez más perfeccionados que reproducen parcialmente dichas capacidades [3].

La IA abarca varios campos de aplicación, por ejemplo [4]:

- **Tratamiento de lenguajes naturales:** Este campo abarca aplicaciones que realicen traducciones entre idiomas, interfaces hombre-máquina que permitan consultar una base de datos o dar órdenes a un sistema operativo que permita una comunicación más amigable y fluida con el usuario.
- **Sistemas expertos:** Aquí se engloban aquellos sistemas dónde la experiencia de personal cualificado se incorpora a dichos sistemas para conseguir deducciones más cercanas a la realidad.
- **Robótica:** Navegación de robots móvil, conducción autónoma, ensamblaje de piezas...
- **Visión Artificial:** Abarca aplicaciones desarrolladas para reconocimiento de objetos, detección de defectos en piezas por medio de la visión o apoyo en diagnósticos médicos.
- **Aprendizaje:** Modelización de conductas para su posterior implantación en computadoras.

Dentro de la IA, un área activa es el *Machine Learning* o aprendizaje automático AA cuyo objetivo es desarrollar técnicas que permitan a las computadoras aprender automáticamente. Más concretamente, se trata de crear aplicaciones capaces de generalizar comportamientos a partir de información suministrada en forma de ejemplos. Se puede definir, por lo tanto, como un proceso de inducción del conocimiento. 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.

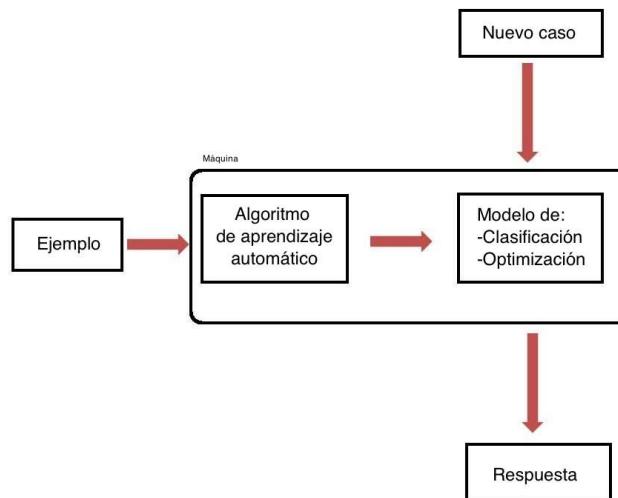


Figura 1.1: Modelo Machine Learning [5].

Machine Learning, se refiere comúnmente a la metodología de aprendizaje automático tradicional, basada en sistemas de decisión lineales y cuyas miles de variantes se llevan utilizando durante años en entornos de computación automatizada.

La experiencia ha demostrado que un sistema que utiliza la técnica de *Machine Learning* tradicional donde el número de variables y el volumen de muestras a analizar es demasiado elevado, perderá de manera exponencial precisión y efectividad, necesitando de una gran cantidad de recursos para mantener unos tiempos de decisión aceptables.

Actualmente, los avances tecnológicos y las mejoras en los algoritmos de AA han hecho que estas técnicas se utilicen en un gran número de aplicaciones, tanto a nivel de negocio como a nivel de investigación. Algunas de las aplicaciones más importantes desarrolladas o pendientes de desarrollar en el futuro basadas en el AA son:

- **Conducción autónoma:** El hecho de que un coche pueda ser dirigido sin la acción directa del ser humano es una de las investigaciones más llamativas en el ámbito de la IA. Los investigadores del ámbito de la automoción emplean el AA para detectar automáticamente objetos tales como señales de stop y semáforos. Además, el AA se utiliza para detectar peatones, lo que contribuye a reducir los accidentes.
- **Automatización industrial:** El AA está ayudando a mejorar la seguridad de los trabajadores en entornos con maquinaria pesada, gracias a la detección automática de personas u objetos cuando se encuentran a una distancia no segura de las máquinas.
- **Electrónica:** El aprendizaje electrónico se usa en la audición automatizada y la traducción del habla. Por ejemplo, los dispositivos de asistencia doméstica que responden a la voz y conocen sus preferencias se basan en aplicaciones de aprendizaje profundo.
- **Investigación médica:** En el ámbito de la salud, como por ejemplo al revisar una mamografía, los algoritmos de AA pueden procesar más información y detectar más patrones que una mente humana. Además, el AA también se puede utilizar para advertir los factores de riesgo de enfermedad en poblaciones grandes.

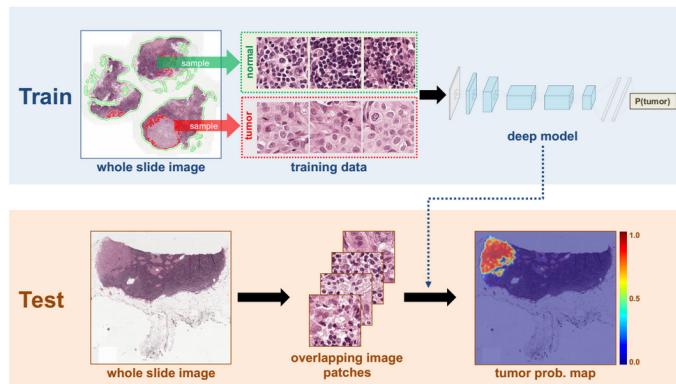


Figura 1.2: Ejemplo de AA aplicado a la detección de cáncer de mama [6].

1.2. Redes neuronales artificiales

Muchas 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.

Por ello, a partir de las redes neuronales biológicas, se crean las redes neuronales artificiales (RNA) las cuales son modelos simplificados de las redes neuronales biológicas. Tratan de imitar 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. En el modelo de una neurona artificial, representado en la figura 1.3, se pueden identificar varios elementos[7].

- **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.

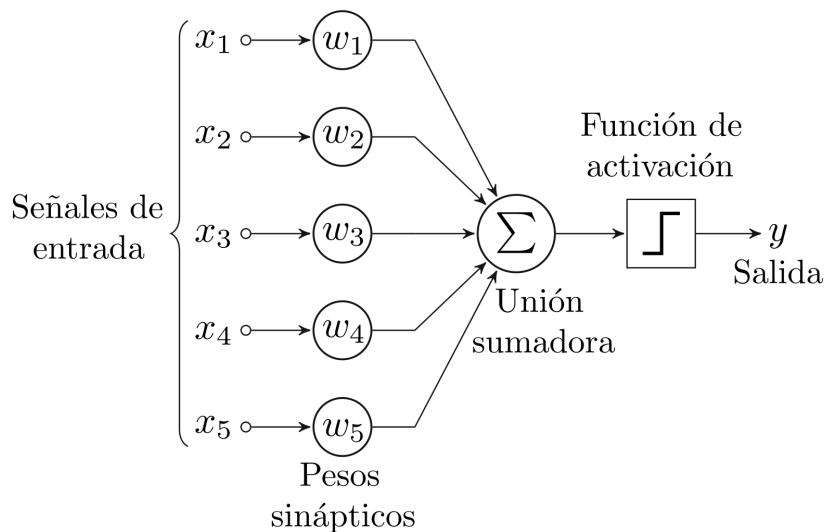


Figura 1.3: Modelo de una neurona artificial [8].

Para definir totalmente una RNA, hay que especificar el conexionado existentes entre cada una de las neuronas que lo forman. Éstas se agrupan en capas, cada una de ellas con un número de neuronas variable y comportamiento similar, constituyendo varias capas de una red neuronal.

Cada capa está conectada total o parcialmente a la que se encuentra inmediatamente después, excepto la última capa, que constituye la salida total de la red neuronal. Existen 3 tipos de capas [9]:

- **Capa de entrada:** También denominada capa sensorial, está compuesta por neuronas que reciben datos o señales procedentes del entorno.
- **Capas ocultas:** Este tipo de capas no tienen conexión directa con el entorno. Proporcionan grados de libertad a la red neuronal gracias a los cuales es capaz de representar de forma más exacta determinadas características del entorno que trata de modelar.
- **Capa de salida:** Está compuesta por neuronas que proporcionan la respuesta de la red neuronal.

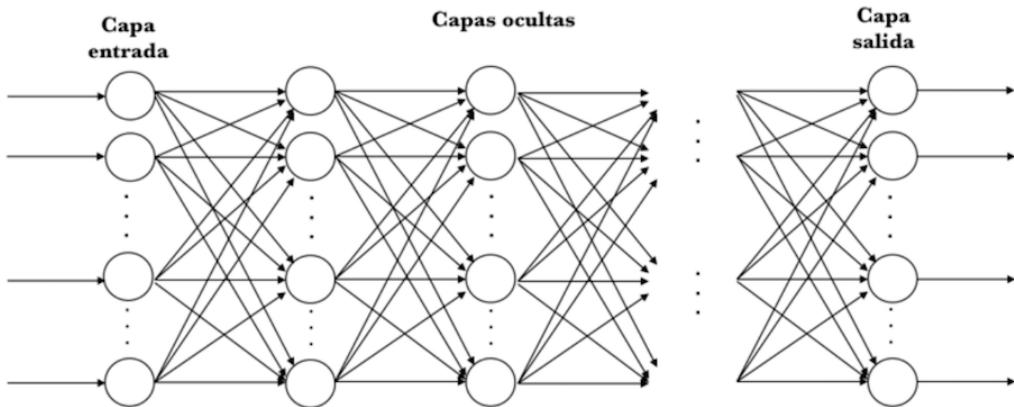


Figura 1.4: Modelo de una red neuronal artificial [10].

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 RNA 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 RNA son capaces de generalizar de casos anteriores a casos nuevos.

1.2.1. Aprendizaje en redes neuronales artificiales

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: supervisado y 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.

1.2.2. Redes neuronales convolucionales (CNN)

Las CNN son uno de los tipos de RNA más utilizadas en el ámbito de la IA, más específicamente del reconocimiento de patrones, como por ejemplo visión artificial o reconocimiento de voz.

La idea básica de las CNN es construir propiedades de invarianza en redes neuronales mediante la creación de modelos invariables para ciertas transformaciones en la entrada.

Para ello, las CNN utilizan una arquitectura especial, mostrada en la figura 1.5 y compuesta de una capa de convolución y otra de submuestreo. La capa de convolución implementa la función de convolución, mientras que en la capa de submuestreo se realiza el *pooling*, que se encarga de generar características invariantes calculando estadísticas de las activaciones de convolución a partir de un pequeño campo receptivo.

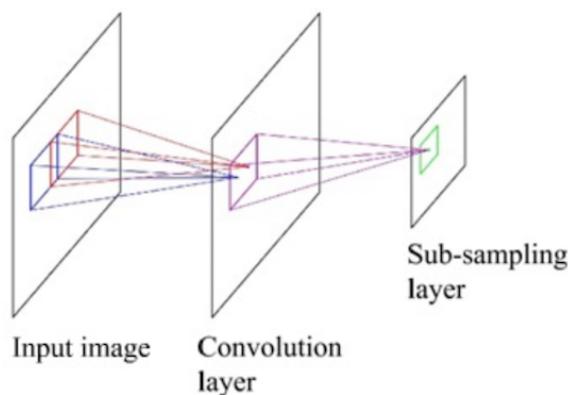


Figura 1.5: Modelo de una red neuronal convolucional [11].

En las CNN, cada neurona de la capa oculta está conectada a un pequeño campo de la capa anterior, el cual será llamado campo receptivo. En la capa de convolución, las neuronas están organizadas en múltiples capas ocultas paralelas, llamadas mapas de características, de tal manera que cada neurona en un mapa de características está conectada a un campo receptivo local. Para cada mapa de características, todas las neuronas comparten el mismo parámetro de peso que se conoce como filtro o kernel [11].

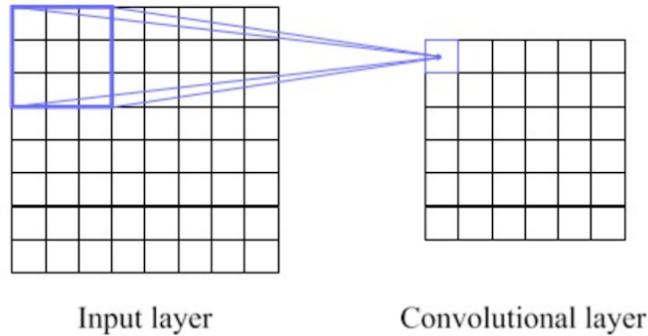


Figura 1.6: Ejemplo de campo receptivo local en la capa convolucional [11].

1.2.3. Entornos software para redes neuronales

Actualmente, el incipiente interés que hay en crear aplicaciones basadas en la inteligencia artificial ha hecho que existan una gran cantidad librerías que permiten desarrollar este tipo de trabajos e investigaciones.

A continuación se definen brevemente algunos de los softwares más utilizados para aplicaciones de aprendizaje máquina en la actualidad.

1. Caffe[12] es uno de los softwares más populares y extendidos actualmente. Este trabajo ha sido desarrollado utilizando este software, por lo que se detallarán sus características principales más adelante.
2. **TensorFlow**[13] es una biblioteca software de código abierto diseñada para el cálculo numérico de alto rendimiento. Su arquitectura flexible permite una fácil implementación de computación en una gran variedad de plataformas, tales como CPUs ó GPUs, y desde escritorios hasta clústeres de servidores y dispositivos móviles

y periféricos. Fue diseñado originalmente por investigadores e ingenieros de Google. Este software cuenta con un sólido soporte para el aprendizaje automático y el aprendizaje profundo y el flexible núcleo de computación numérica del que dispone hace que puede ser utilizado en muchos otros dominios científicos.

3. **Keras**[14] es una API de redes neuronales de alto nivel escrita en Python y capaz de ejecutarse sobre TensorFlow, CNTK o Theano. Fue desarrollada con el objetivo principal de permitir una experimentación rápida. El enfoque de este software es que el poder pasar de la idea al resultado en el menor tiempo posible es la clave para hacer una buena experimentación.

Las características y ventajas principales que pueden definir a esta librería son:

- **Facilidad de uso:** Keras es una API diseñada para seres humanos, no para máquinas. Sigue las mejores prácticas para reducir la carga cognitiva: ofrece APIs consistentes y simples, las cuales minimizan el número de acciones de usuario requeridas para casos de uso común y proporciona comentarios claros y procesables ante el error del usuario.
- **Modularidad:** Un modelo se entiende como una secuencia o un gráfico de módulos independientes y completamente configurables que se pueden conectar con la menor cantidad de restricciones posible. En particular, las capas neuronales, las funciones de costos, los optimizadores, los esquemas de inicialización, las funciones de activación y los esquemas de regularización se tratan de módulos independientes que se puede combinar para crear nuevos modelos.
- **Facilmente extensible:** Los nuevos módulos son simples de agregar, ya sea como nuevas clases o como funciones. Al crearlos se permite una total expresividad, lo que hace de Keras una librería altamente adecuada para la investigación avanzada.
- **Trabaja con Python:** No hay archivos de configuración de modelos definidos en un formato declarativo. Los modelos se escriben en código de Python, el cual permite fácilmente la extensión y es compacto y fácil de depurar.

4. **Darknet**[15]: Se trata de un framework de código abierto de redes neuronales

escrito en C y CUDA. En su página web oficial se pueden observar varios de los trabajos desarrollados por sus creadores. Algunos de los más destacados son su componente detector en tiempo real, utilizando para el entrenamiento de la red la base de datos COCO, la cual se utilizará en este trabajo, y su clasificador de imágenes para el reto 1000-class ImageNet.

1.2.4. Deep Learning

El *Deep Learning* o aprendizaje profundo (AP) es un conjunto de algoritmos que pertenecen a la clase AA que intentan modelar abstracciones de alto nivel usando arquitecturas compuestas de transformaciones no lineales múltiples.

Las redes neuronales artificiales son un paradigma de aprendizaje automático inspirado en las neuronas de los sistemas nerviosos biológicos. La mayor parte de los métodos de aprendizaje emplean arquitecturas de redes neuronales, por lo que, a menudo, los modelos de aprendizaje profundo se denominan redes neuronales profundas. Unas de las RNA más utilizadas en el AP son las CNN, explicadas en el apartado 1.2.2. En el contexto de la detección de imágenes, que es donde se encuadra este TFG, las CNN resultan idóneas ya que con ésta técnica se procesa cada imagen por secciones en lugar de tener un parámetro por cada píxel de ésta.

El término profundo suele hacer referencia al número de capas ocultas en la red neuronal. Las redes neuronales tradicionales solo contienen dos o tres capas ocultas, mientras que las redes profundas pueden tener hasta 150.

La gran precisión que se consigue utilizando técnicas de *Deep Learning* contribuye a que se satisfagan las expectativas de los usuarios, y resulta crucial en aplicaciones críticas para la seguridad, tales como los vehículos sin conductor. Los últimos avances han llegado a un punto en el que este supera a las personas en algunas tareas, como por ejemplo, en la clasificación de objetos presentes en imágenes, lo cual tiene una gran importancia en el ámbito de la medicina, ya que con estas técnicas se pueden detectar, por ejemplo en radiografías, elementos que habían sido obviados por el ser humano.

El AP es una forma especializada de AA. Un flujo de trabajo de AA comienza por la

1.2. REDES NEURONALES ARTIFICIALES

extracción manual de las características relevantes de la imagen. Estas características se utilizan entonces para crear un modelo que categoriza los objetos de la imagen. Con un flujo de trabajo de AP, las características relevantes se extraen directamente de la imagen.

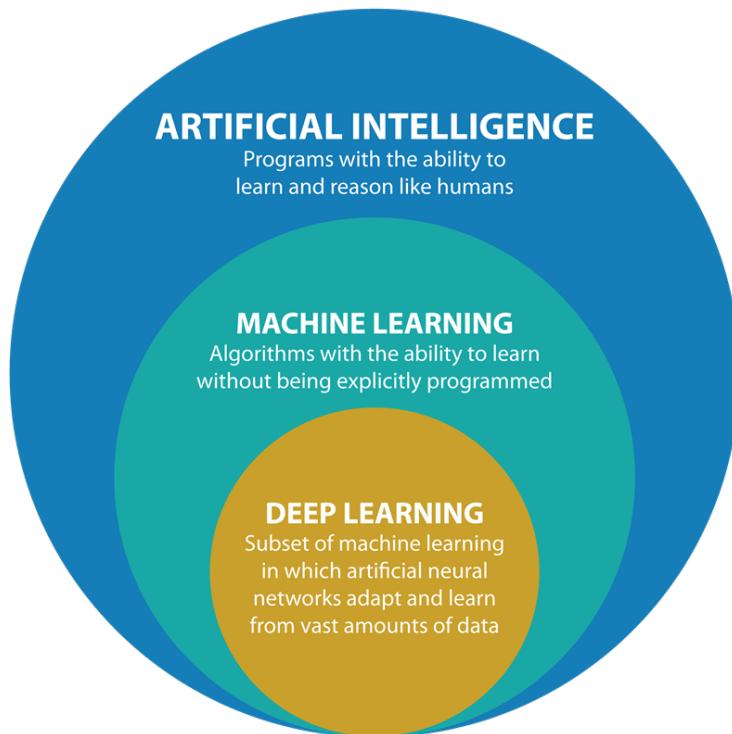


Figura 1.7: Marco de la Inteligencia Artificial [16].

Otra diferencia clave es que con los algoritmos de AP la escala aumenta con los datos, mientras que, en el caso del aprendizaje superficial, existe convergencia. El aprendizaje superficial hace referencia a los métodos de que llegan a un punto muerto en cierto nivel de rendimiento cuando se agregan más ejemplos y datos de entrenamiento a la red.

Una ventaja fundamental de las redes de aprendizaje profundo es que suelen seguir mejorando a medida que aumenta el tamaño de los datos.

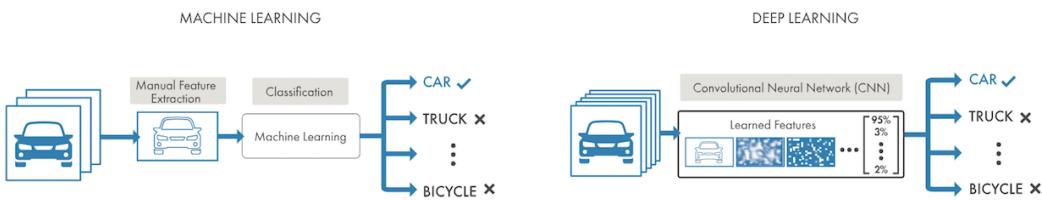


Figura 1.8: Comparación de un enfoque de aprendizaje automático para la categorización de vehículos (izquierda) con el aprendizaje profundo (derecha) [17].

1.3. Antecedentes en Robotics de la URJC

Actualmente, en en Robotics de la URJC ya se han desarrollado varios trabajos basados en la inteligencia artificial y más concretamente en *Deep Learning*. En ellos se han utilizado algunos de los entornos mencionados anteriormente para entrenar una o varias redes neuronales. Tras ello, se han abordado tareas de clasificación y detección de objetos analizando tras ellos la capacidad que tenían las redes neuronales ya entrenadas para realizar ese cometido.

Nuria Oyaga¹ [18] realizó un trabajo basado en la clasificación de dígitos manuscritos. Para el estudio y utilización de las redes neuronales usó la plataforma *Caffe*, la cual provee de una red básica ideal para comenzar en el campo de la clasificación.

Como banco de imágenes de entrada, cuyo objetivo es entrenar la red neuronal, utilizó Modified National Institute of Standards and Technology (MNIST). Esta base de datos está constituida por números escritos a mano y consta de un conjunto de entrenamiento de 60.000 ejemplos y otro de prueba de 10.000. Estas imágenes están dispuestas en escala de grises.

¹<https://jderobot.org/Noyaga-tfg>



Figura 1.9: (a) Muestras de la base de datos MNIST. (b) Muestras de imágenes aplicando el filtro Sobel.

Para evaluar el rendimiento del clasificador, modificó la base de datos MNIST realizando transformaciones sobre sus imágenes y utilizándolas posteriormente para entrenar la red neuronal. Algunas de estas transformaciones fueron la inclusión de filtros a los ejemplos, tales como el Laplaciano, Sobel o Canny y modificaciones basadas en el escalado, translación y rotación de las imágenes de entrada.

Además de la evaluación de prestaciones de la red, Nuria realizó un componente para la clasificación de dígitos manuscritos en tiempo real. En el presente trabajo se ha realizado una aproximación de este componente pero con la diferencia de que el objetivo de éste será la detección de objetos.

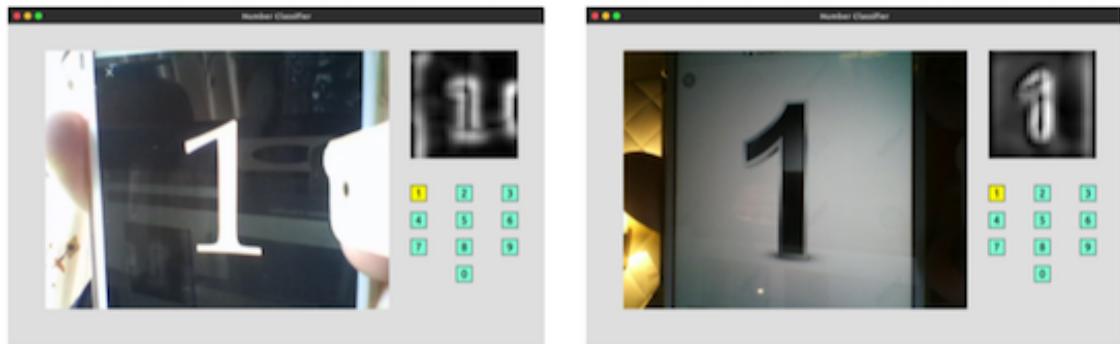


Figura 1.10: Ejemplo del componente clasificador.

Por otra parte, David Pascual² [19], realizó un trabajo basado en la clasificación de dígitos manuscritos utilizando la plataforma Keras. Para ello, emplea un modelo proporcionado por éste software y lo entrena utilizando la base de datos MNIST. Además, evalúa

²<https://jderobot.org/Dpascual-tfg>

las prestaciones de la red realizando una serie de transformaciones sobre los ejemplos de los dígitos y crea un componente para la clasificación de números en tiempo real.

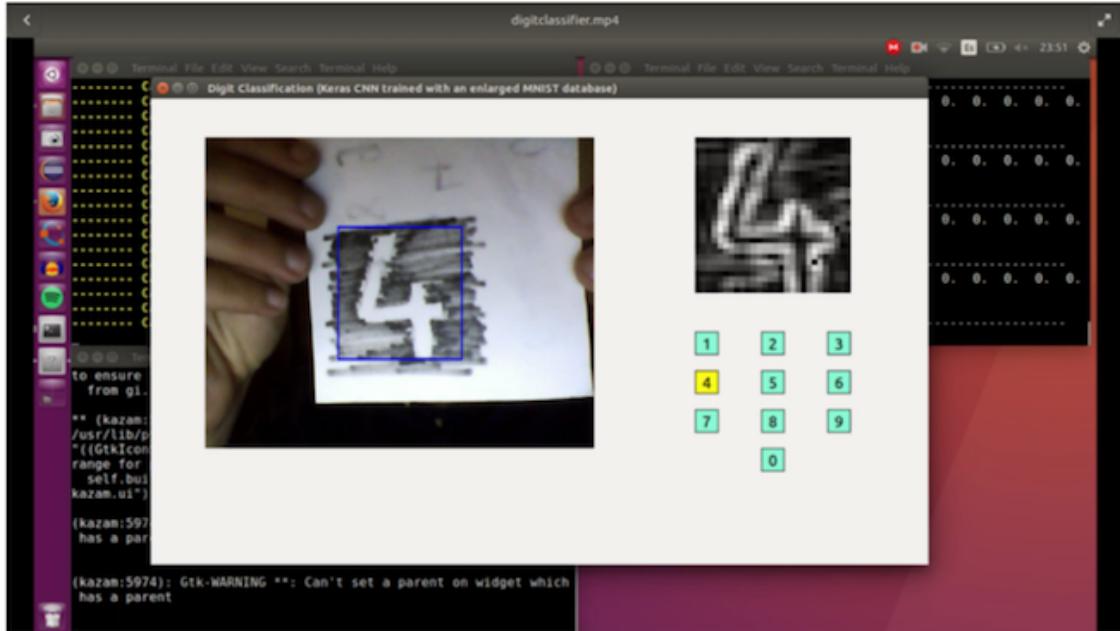


Figura 1.11: Ejemplo del componente clasificador.

En lo referente a la detección de objetos, en Robotics de la URJC también se ha desarrollado la herramienta *Detection Suite*³ que se trata de un conjunto de utilidades que simplifican la evaluación de los conjuntos de datos de detección de objetos más comunes con varias redes neuronales de detección de objetos.

La idea principal es ofrecer una infraestructura genérica para evaluar los algoritmos de detección de objetos contra un conjunto de datos y calcular las estadísticas más comunes, tales como *Intersection Over Union*, *precision* y *recall*. Esta herramienta acepta un gran número de bases de datos internacionales como PascalVOC y permite la comparación de varias arquitecturas de *Deep Learning* utilizando exactamente la misma prueba.

³<https://github.com/JdeRobot/dl-DetectionSuite>

1.3. ANTECEDENTES EN ROBOTICS DE LA URJC

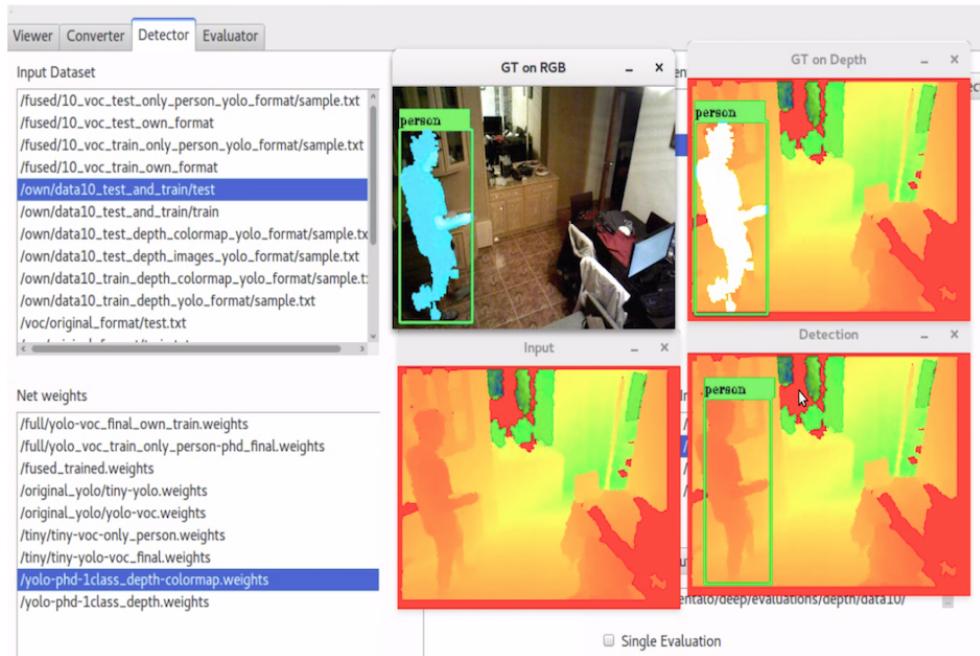


Figura 1.12: Herramienta DetectionSuite.

Además, Nacho Condés⁴ [20] ha desarrollado un componente de detección de objetos en tiempo real llamado *dl-objectdetector*⁵. Este componente soporta tanto *Keras* como *TensorFlow* y está escrito en *Python*.

⁴<https://jderobot.org/Naxvm-tfg>

⁵<https://github.com/JdeRobot/dl-objectdetector>

Capítulo 2

Objetivos y metodología

2.1. Objetivos

El objetivo principal de este proyecto es el entendimiento de las redes neuronales y el Aprendizaje Profundo o *Deep Learning* aplicado a la detección visual de objetos. Este objetivo principal se ha articulado en varios subobjetivos.

- **Diseño y desarrollo de una aplicación que detecte objetos en imágenes en tiempo real sobre flujos de video:** Estará integrada con el entorno Caffe y a partir de los pesos y la estructura de redes neuronales entrenadas o preentrenadas procesará las imágenes de entrada captadas por una cámara de vídeo y realizará la detección de objetos sobre ellas.
- **Entrenamiento de una red neuronal propia para la detección de objetos:** Utilizando Caffe, se definirán todos los componente que provee este entorno para entrenar una red neuronal. Además, se creará una base de datos personalizada para utilizarla como conjunto de datos para el entrenamiento del modelo y para su evaluación (test).
- **Evaluación y comparación cuantitativa de la red propia con otras preentrenadas:** Se realizarán diferentes experimentos para evaluar y comparar las prestaciones de la red entrenada y las preentrenadas a partir de diferentes métricas.

2.2. Metodología y plan de trabajo

Durante este trabajo fin de grado se han utilizado varias herramientas que han contribuido a que la dinámica de desarrollo fuera organizada y cooperativa entre todas las personas involucradas y a que en todo momento estuvieran al tanto de todo el material que se iba creando y redactando.

La principal herramienta metodológica han sido las reuniones concertadas periódicamente con los tutores. En éstas, se exponían los avances realizados durante la semana, las dudas surgidas y se exponían y proponían nuevas líneas para el desarrollo del trabajo.

Para la organización del código generado se ha utilizado el repositorio *GitHub*¹. En él se encuentra todo el *software* desarrollado durante el trabajo, con el objetivo de que pueda ser reutilizado, corregido o modificado en función de las necesidades.

Con el propósito de documentar, semana a semana, todo lo desarrollado, se ha utilizado la *Wiki*² que proporciona JdeRobot. Gracias a esto se podía tener acceso al trabajo realizado semanas atrás y los tutores podían revisar antes de la reunión lo realizado durante la semana, haciendo que ésta fuera más práctica y fluida.

Como punto de partida para conseguir los objetivos comentados en el apartado 2.1, se planificaron varias fases de estudio de las diferentes herramientas y conceptos y fases de desarrollo:

- **Estudio de redes neuronales artificiales**, de sus características, de su estructura y del funcionamiento y tipos de entrenamiento. Tras esto se hace un estudio particular de las redes neuronales convolucionales o CNN y del *Deep Learning*, dos de los conceptos utilizados durante el trabajo. Además, se realiza un estudio de la plataforma usada para aplicarlos, es decir, de Caffe, de sus características principales y de sus funcionalidades. Se completan algunos de los tutoriales que proveen sus creadores como primer paso para conocer su utilización y capacidad.
- **Programación de una aplicación** que usa redes de Caffe para detectar objetos

¹<https://github.com/RoboticsURJC-students/2016-tfg-David-Butragueno>

²<https://jderobot.org/Davidbutra-tfg>

en flujos de video en tiempo real.

- **Estudio de varias bases de datos aplicadas a la detección,** la distribución de los diferentes objetos dentro de sus imágenes y el formato de sus ficheros de anotaciones. Además se estudian varias métricas para la evaluación de detectores visuales que se utilizan en este trabajo. De esta manera se facilita el análisis de los datos obtenidos tras evaluar la red entrenada y las preentrenadas y así poder realizar una mejor comparación de sus prestaciones
- **Estudio de la técnica SSD:** Se realiza una lectura exhaustiva de la documentación realizada por los creadores de esta técnica [21] para conocer sus principales características, los resultados de sus experimentos realizados y las conclusiones obtenidas a partir de ellos. Tras esto se entrena un modelo SSD utilizando la plataforma Caffe y una combinación de las bases de datos estudiadas anteriormente como conjunto de datos de entrenamiento.
- **Evaluación cuantitativa y comparativa** entre redes preentrenadas y la red propia para la detección visual de objetos a partir de las métricas que calcula la herramienta *Detection Suite*.

Capítulo 3

Infraestructura utilizada

3.1. Plataforma Caffe

Caffe[12] es un entorno 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.

Los puntos que definen a este entorno y que lo posicionan frente a otros orientados al mismo objetivo son los siguientes:

- La arquitectura expresiva de Caffe fomenta la aplicación y la innovación. Los modelos y la optimización se definen por configuración y es posible cambiar del uso entre CPU y GPU modificando un único indicador.
- El código extensible fomenta el desarrollo activo. En el primer año de Caffe, más de 1000 desarrolladores han participado en el proyecto y ha tenido muchos cambios significativos. Gracias a estos colaboradores, el entorno sigue estando a la última en el estado del arte tanto en código como en modelos.
- La velocidad de ejecución hace que Caffe sea perfecto para experimentos de investigación e implementación en la industria. Caffe puede procesar más de 60 millones de imágenes al día con una sola GPU *NVIDIA K40**. Esto significa 4 ms/imagen para el aprendizaje, siendo las versiones más recientes de la biblioteca y el hardware aún más rápidos. Por todo esto, Caffe es una de las plataformas más rápidas disponibles.

3.1. PLATAFORMA CAFFE

- Comunidad: Caffe impulsa proyectos de investigación académica e incluso aplicaciones industriales a gran escala en visión, voz y multimedia. Todo el código desarrollado por la comunidad se encuentra disponible en su repositorio de *GitHub*¹.

3.1.1. Interfaz vía comandos

Caffe dispone de una interfaz de línea de comandos llamada *cmdcaffe*[22] la cuál es la herramienta utilizada 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, reanudar el aprendizaje a partir de *snapshots* guardadas y ajustar los modelos a los nuevos datos y tareas.
- **Test:** El comando *caffe test* califica los modelos ejecutándolos en la fase de test e informa de la salida de la red como su puntuación. En primer lugar se informa de la puntuación por lotes de datos de entrada y finalmente el promedio general.
- **Análisis comparativo:** El comando *caffe time* compara el modelo capa a capa. Esto es útil para comprobar el rendimiento del sistema y medir los tiempos de ejecución relativos a los modelos.
- **Diagnósticos:** El comando *caffe device_query* informa de los detalles de la GPU para referencia y comprobación del identificador de los dispositivos disponibles para ser ejecutados en máquinas *multi-GPU*.
- **Paralelismo:** El indicador *-gpu* de la herramienta Caffe provee una lista separada por comas de IDs para ejecutar procesos en múltiples GPUs.

3.1.2. Interfaz con lenguaje Python

La interfaz de Python *pycaffe*[22] 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, 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 esta interfaz son:

¹<https://github.com/BVLC/caffe/>

- **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** es la interfaz de resolución.
- **caffe.io** maneja funciones de entrada/salida.
- **caffe.draw** visualiza las arquitecturas de red.

3.1.3. 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).

Las capas de Caffe y sus parámetros están especificados en la definición del buffer de protocolo para el proyecto, en el archivo `caffe.proto`². Además, existe documentación[23] donde se pueden examinar las características principales de los diferentes tipos de capas en Caffe, así como de los subtipos de cada una de ellas.

3.1.3.1. Capas de datos

Los datos entran en Caffe a través de las capas de datos, las cuales se encuentran en la parte inferior de las redes. Estos datos pueden proceder 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.

Este tipo de capas son las encargadas de realizar tareas comunes de preprocesamiento de los datos de entrada, tales como sustracción media o escalado, especificando para ello los parámetros de transformación de la propia capa.

Todos los sub-tipos de este tipo de capa se especifican a continuación:

- **ImageData:** Lee imágenes sin procesar.
- **Database:** Lee los datos de ficheros LevelDB o LMDB.

²<https://github.com/BVLC/caffe/blob/master/src/caffe/proto/caffe.proto>

- **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 están siendo desplegadas.
- **Memory Data:** Lee archivos directamente desde memoria.
- **Dummy Data:** Utilizado para datos estáticos.

3.1.3.2. 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 de color si se trata de una imagen en escala de grises o 3 canales de color 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.

Existen varios tipos de capas de visión, los más comunes son:

- **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 diferentes 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.

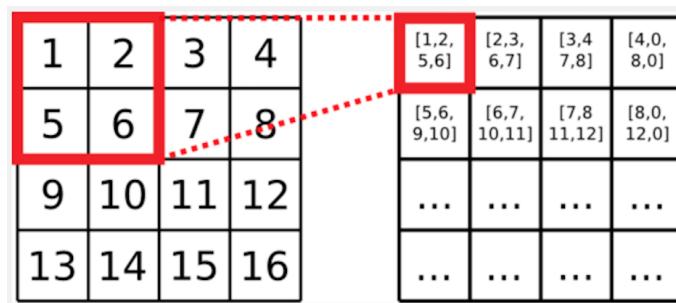


Figura 3.1: Ejemplo convolución con tamaño del núcleo 2x2.

- **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 $m \times n$. A continuación, se aplica una función de agrupación para cada uno de los bloques. Tras este proceso, se obtendrá una matriz de características más pequeña. 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.

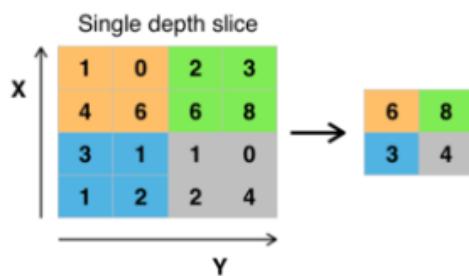


Figura 3.2: *Pooling* con función de agrupación del máximo.

3.1.3.3. Capas comunes

- **Inner Product:** Este tipo de capa trata la entrada como un vector simple y produce una salida en forma de otro vector, estableciendo la altura y la anchura de cada *blob* a 1.
- **Dropout:** Las capas Dropout son las encargadas de abordar el problema del sobreentrenamiento en las redes neuronales. El concepto central es que cada capa oculta de la red neuronal entrenada con Dropout debe aprender a trabajar con muestras de otras capas elegidas aleatoriamente. Esto hace que cada capa oculta de la red sea más robusta y sea capaz de crear características útiles por sí misma sin depender de otras capas ocultas para corregir sus errores, además de evitar que exista una gran co-adaptación entre cada una de ellas.

3.1.3.4. Capas de pérdida

Estas capas son las encargadas de propagar la función de pérdida durante el entrenamiento de la red neuronal. Esta función de pérdida toma las predicciones que realiza el modelo y los valores objetivos, es decir, los que realmente se espera que la red produzca, y calcula cuán lejos se está de ese valor, comprobando de esta manera cómo funciona la red para un ejemplo específico. Esta función ajusta los pesos de la red reduciendo la pérdida del modelo para cada ejemplo.

En Caffe, la pérdida se calcula con la propagación hacia adelante de la red. Cada capa toma un conjunto de datos de entrada y produce unos datos de salida. Algunas de las salidas de estas capas pueden utilizarse para calcular la pérdida del modelo. Algunas de las capas de pérdida definidas en Caffe son las siguientes:

- **Softmax with Loss:** Esta capa toma un vector *N-dimensional* de números reales y lo transforma en otro vector de números reales dentro del rango (0,1).

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^N e_k^a} \quad (3.1)$$

- **Sum-of-Squares / Euclidean:** Calcula la suma de cuadrados de diferencias de sus dos entradas.
- **Sigmoid Cross-Entropy Loss:** Calcula las pérdidas sigmoidales cruzadas, a menudo utilizadas para predecir objetivos interpretados como probabilidades.
- **Accuracy / Top-k layer:** Dado un ejemplo específico de entrada en la red neuronal, califica la salida de la red con respecto a lo que se esperaba que fuera la salida de ésta.

3.1.3.5. Capas de activación

Estas capas son las encargadas de ejecutar la función de activación sobre las diferentes capas de la red neuronal. En general, estas capas son operandos que toman un dato de la salida de la capa anterior y generan datos con las mismas dimensiones ejecutando la función de activación que se haya definido. Algunas de las más comunes son las siguientes:

- **Rectified Linear Unit (ReLU):** Se trata de una función lineal rectilínea con pendiente uniforme. La salida de esta función es 0 si la entrada es menor que 0 y

la salida bruta en caso contrario, es decir, si la entrada es mayor que 0, la salida es igual a la entrada.

$$f(x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases} \quad (3.2)$$

Esta función es no lineal y tiene la ventaja de no tener errores en la propagación hacia atrás durante el entrenamiento de la red. Gráficamente, se puede representar como muestra la Figura 3.3.

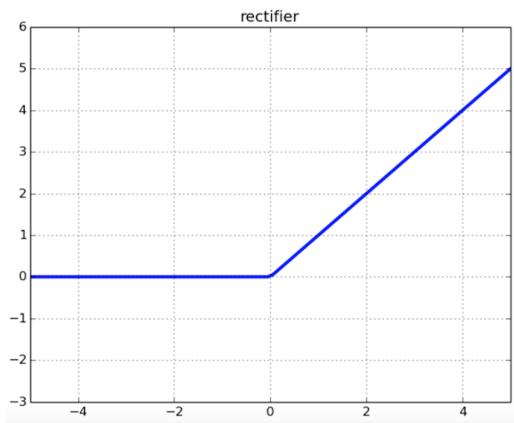


Figura 3.3: Función de activación ReLU.

- **Sigmoid:** Muchos procesos de aprendizaje automático, e incluso sucesos naturales muestran una aceleración considerable intermedia al cambio de estado con períodos de adaptación entre estados, esta evolución es descrita por la función sigmoide:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (3.3)$$

A menudo la función sigmoide se refiere al caso particular de la función sigmoidal, que tiene una típica forma de "S". Gráficamente, se puede representar como se muestra en la Figura 3.4.

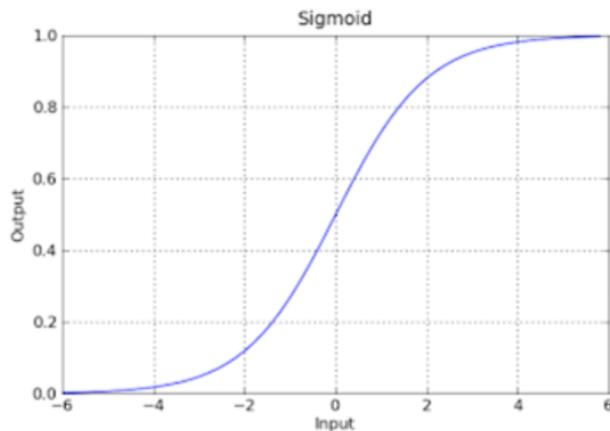


Figura 3.4: Función de activación Sigmoide.

3.2. Organización JdeRobot

JdeRobot[24] es una organización creada para desarrollar aplicaciones en robótica y visión artificial. Este campo de acción incluye la utilización de sensores, como por ejemplo cámaras, actuadores y software inteligente. El *software* de JdeRobot está escrito principalmente utilizando los lenguajes C++ y Python proporcionando un entorno de programación distribuido basado en componentes donde el programa de aplicación está compuesto por una colección de varios componentes asíncronos concurrentes. Estos elementos pueden ejecutarse en diferentes equipos y están conectados mediante el *middleware* de comunicaciones ICE (Internet Communication Engine) o mediante mensajes ROS (Robot Operating System). Los componentes interactúan a través de interfaces explícitas. Cada uno de ellos puede tener su propia interfaz gráfica de usuario independiente o ninguna.

JdeRobot simplifica el acceso a dispositivos hardware desde el programa de aplicación. Obtener mediciones de sensores se logra con llamadas a funciones locales. La plataforma conecta las llamadas al *driver* de los componentes los cuales están conectados a los sensores o actuadores, que pueden ser reales o simulados y remotos o locales. Estas funciones construyen la API para la capa de abstracción de hardware. Los robots y sensores actualmente soportados son:

- Sensores RGBD: Kinect and Kinect2 de Microsoft, Asus Xtion.
- Robots con ruedas: 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, como mpeg o avi, y cámaras IP, como Axis.

JdeRobot incluye varias herramientas y librerías de programación de robots, las cuales se especifican a continuación:

- Teleoperadores para varios robots, viendo sus sensores y dirigiendo sus motores. Algunos de ellos están basados en la web y funcionan en *smartphones*.
- La herramienta *VisualStates* para programar el comportamiento de robots con autómatas de estado finito.
- La herramienta *Scratch2JdeRobot* para la programación de robots, incluyendo drones, con lenguaje gráfico estándar.
- Una herramienta para ajustar los filtros de color.

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

3.2.1. Servidor de imágenes CameraServer

Se trata de un componente facilitado por JdeRobot que actúa como servidor de imágenes, proporcionando a las aplicaciones de estas imágenes procedentes de un número determinado de cámaras, ya sean reales o simuladas, o de un archivo de vídeo.

Para el uso correcto de este componente, es necesario editar un fichero de configuración en función de las necesidades concretas de la máquina. En este archivo, con extensión *cfg*, se pueden definir varios parámetros:

- *EndPoint* del servidor que va a recibir la petición.
- Número de cámaras que se usarán.

3.2. ORGANIZACIÓN JDEROBOT

- Configuración de las cámaras. Se podrán modificar los siguientes parámetros relacionados con las cámaras:
 - Nombre de la cámara.
 - El parámetro URI que define la fuente de vídeo.
 - Numerador y denominador de *frame rate*
 - Formato de la imagen.
 - Altura y anchura de las imágenes.

3.2.2. DetectionSuite

*Detection Suite*³ es un conjunto de herramientas que simplifica la evaluación de los conjuntos de datos de detección de objetos más comunes utilizando redes neuronales entrenadas para ese fin.

La idea principal es ofrecer una infraestructura genérica para evaluar los algoritmos de detección de objetos contra un conjunto de datos y calcular las estadísticas más comunes, como por ejemplo, *Intersection Over Union*, Precisión y Recall.

Los sistemas y plataformas con las que está integrado *Detection Suite* son:

- Sistemas operativos: Linux y MacOs.
- Formatos de bases de datos: YOLO, COCO, ImageNet, PascalVOC, Jderobot Recorder Logs, Princeton RGB, Spinello.
- Plataformas de detección de objetos: YOLO (Darknet), TensorFlow, Keras, Caffe, Background subtraction.
- Entradas compatibles para la implementación de redes: Cámara Web/Cámara USB, Videos, Vídeos de ICE (Internet Communication Engine), Vídeos de ROS (Robot Operating System), JdeRobot Recorder Logs.

Detection Suite provee varios módulos⁴ con los cuáles se pueden realizar diferentes funcionalidades:

³<https://github.com/JdeRobot/DetectionSuite>

⁴<https://github.com/JdeRobot/DetectionSuite/wiki>

- **Auto Evaluator:** Herramienta ejecutada por línea de coman dos que evalúa múltiples redes en uno o varios conjunto de datos en una sola ejecución.
- **Viewer:** Herramienta gráfica que lee las imágenes y anotaciones de un conjunto de datos para etiquetarlas con sus respectivos nombres de clases, mostrándolos por pantalla.
- **Converter:** Transforma conjuntos de datos de entrada a varios formatos de bases de datos.
- **Detector:** Herramienta gráfica que crea una base de datos con las detecciones obtenidas.
- **Evaluator:** Misma funcionalidad que la herramienta *Auto Evaluator* pero desde la interfaz gráfica.
- **Deployer:** Realiza la detección de objetos, utilizando TensorFlow, Keras, Darknet o Caffe, sobre vídeos grabados o imágenes captadas por una cámara en tiempo real.

3.3. Bases de datos de detección visual

Para entrenar un sistema que sea capaz de reconocer ciertos objetos dada una imagen de entrada es necesario contar con un gran conjunto de datos. Dentro de estos conjuntos, es necesario que haya ejemplos destinados al entrenamiento de la red, otros a la validación de ésta y otros a las pruebas o *test* de las prestaciones de ésta. Además de estos ejemplos, deben existir unos ficheros denominados anotaciones dónde se especifican las coordenadas y etiqueta verdadera de los objetos de cada una de las imágenes pertenecientes a los conjuntos de entrenamiento y validación.

Durante este trabajo, se han utilizado dos bases de datos diferentes. A continuación se presentan sus principales características, la cantidad de imágenes que hay en cada uno de los conjuntos y la estructura y tipo de fichero en la que se definen las anotaciones.

3.3.1. PASCAL Visual Object Classes

PASCAL Visual Object Classes[25], o PascalVOC, es una base de datos que contiene 11.530 imágenes de entrenamiento y validación que representan 27.450 objetos diferentes

3.3. BASES DE DATOS DE DETECCIÓN VISUAL

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. La tarea de detección consiste en estimar 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 y validación. Específicamente, la distribución de objetos para la tarea de detección se muestra en la Figura 3.5.

	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 3.5: Distribución de objetos en PascalVOC [26].

En cuanto a los archivos de anotaciones, éstos 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* e *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>
```

3.3.2. Common Objects in Context

Common Objects in Context[27], conocido como COCO por sus siglas, es un conjunto de datos creado para la detección y segmentación visual 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 categorías 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 en la Figura 3.6:

```
{  
    "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 3.6: Estructura básica de anotaciones en COCO [28].

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, en cuyo caso se usan polígonos, o una colección de objetos, en cuyo caso se usa Run-Length Encoding (RLE) que es una forma de compresión de datos. Hay que tener en cuenta que un solo objeto puede requerir múltiples polígonos. Las anotaciones de multitudes se utilizan para etiquetar grandes grupos de objetos, como 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 3.7: Estructura básica de anotaciones en COCO [28].

En referencia a los datos, el conjunto de datos 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 270.000 personas segmentadas y un total de 886.000 instancias de objetos segmentados en los datos de entrenamiento y validación. La versión acumulada de 2015 contiene 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:

3.3. BASES DE DATOS DE DETECCIÓN VISUAL

	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 3.8: Distribución de objetos en COCO.

Capítulo 4

Detección visual de objetos con Single Shot MultiBox y Caffe

Uno de los objetivos de este trabajo fin de grado es el desarrollo de una aplicación que permita el uso de redes SSD para detección visual de objetos en flujos de vídeo.

En este capítulo se explica todo lo necesario para desarrollar y utilizar esta aplicación. Se comienza con una introducción a la técnica SSD, tanto a la estructura del modelo como a las características principales del entrenamiento de éste. Tras esto se continua con la explicación de todos los módulos que forman parte de la aplicación y cómo se ejecuta ésta. Después se muestra como se integra Caffe con SSD y finalmente se explica como entrenar una red propia usando la plataforma Caffe, la cuál podrá ser usada en la aplicación desarrollada.

4.1. Técnica Single Shot MultiBox Detection

Single Shot MultiBox Detection[21], o SSD por sus siglas, es un modelo diseñado para detectar objetos en imágenes utilizando para ello únicamente una red neuronal profunda. Fue lanzado a principios de diciembre de 2015 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 fotogramas 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 [30]:

- **Single Shot:** Hace referencia a que las 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* que es utilizada como referencia [32].
- **Detector:** La red neuronal se trata de un detector de objetos que también se encarga de clasificarlos.

En el momento de la predicción, la red neuronal genera las puntuaciones para cada categoría de objetos en cada cuadro predeterminado y produce modificaciones 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.

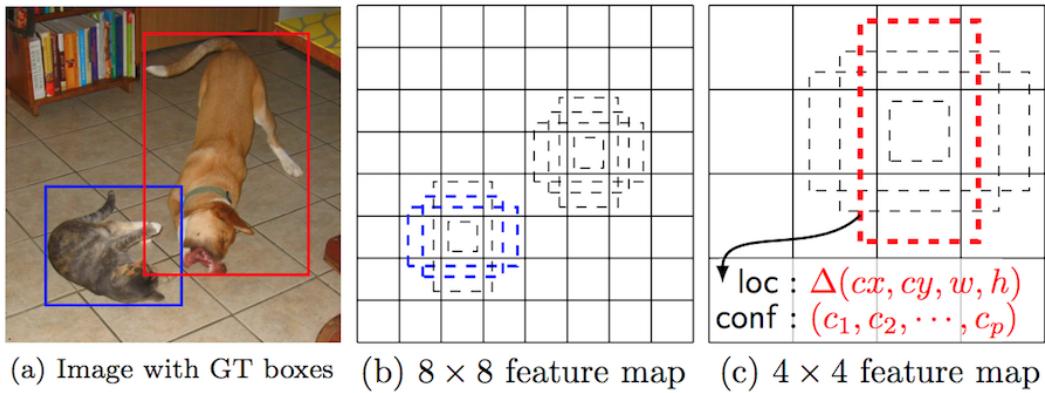


Figura 4.1: Framework SSD [21].

SSD solo necesita una imagen de entrada y los cuadros delimitadores reales para cada objeto durante el entrenamiento (Figura 4.1 (a)). 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 8×8 y 4×4 en (b) y (c)). Para cada cuadro predeterminado se predicen tanto los desplazamientos de forma como la confianza para todas las categorías de objetos ((c_1, c_2, \dots, c_p)). En el momento del entrenamiento, se comparan estos cuadros predeterminados con los cuadros reales.

4.1.1. 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[31] 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 estructura 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 4.2. Para una capa de características de tamaño $m \times n$ con p canales, el elemento básico para predecir los parámetros de una posible detección es un núcleo pequeño de $3 \times 3 \times p$ el cual 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.

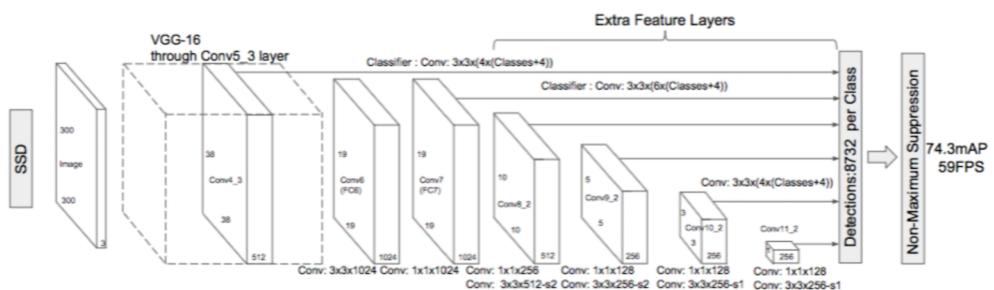


Figura 4.2: Arquitectura SSD [21].

- **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 $(c + 4)kmn$ salidas para un mapa de $m \times n$. 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.

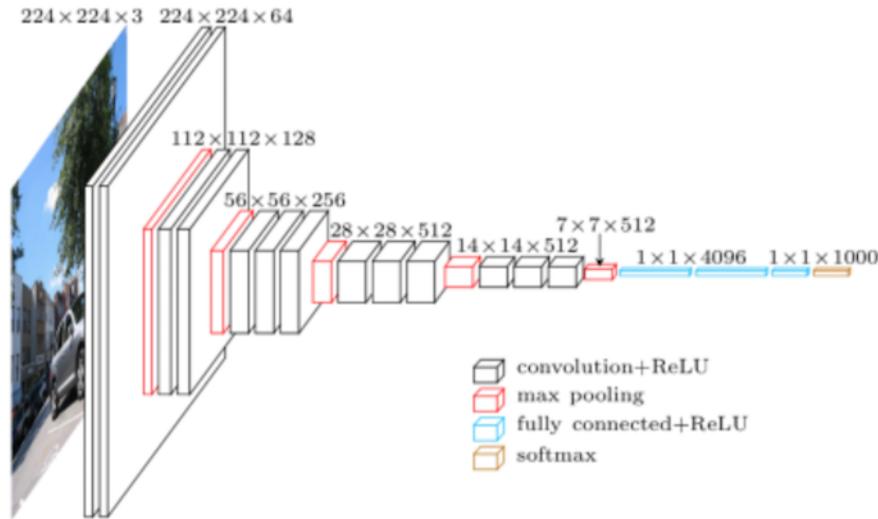


Figura 4.3: Arquitectura VGG-16 [30].

4.1.2. 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 los 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.

- **Estrategia de emparejamiento:** Durante el entrenamiento es necesario determinar qué cuadros predeterminados corresponden a la *bounding box* real de un objeto determinado y entrenar la red en función de esto. En primer lugar, se emparejan cada cuadro real con el predeterminado cuya superposición genere un valor mayor de *Intersection over Union*, IoU, tal y como se realiza en *Multibox*[32]. Posteriormente, y a diferencia de la técnica *MultiBox*, se empareja cada cuadro real con cada caja predeterminada con la que produzca un índice Jaccard superior a 0,5. De esta manera se simplifica el problema del aprendizaje, permitiendo a la red predecir coeficientes de predicción altos para múltiples cajas predeterminadas que se superponen, en lugar de requerir que se escoja únicamente la que tiene un parámetro IoU máximo.
- **Objetivo del entrenamiento:** El objetivo del entrenamiento de SSD es derivado del de MultiBox pero se extiende para manejar múltiples categorías de objetos. La función de pérdida objetivo es una suma ponderada de la pérdida por localización y la pérdida por confianza. Por otra parte, se calcula utilizando la función *Softmax*, explicada en el apartado 3.1.3.4, sobre las confianzas de múltiples clases.
- **Escoger escalas y relaciones de aspecto de los cuadros predeterminados:** Para manejar diferentes escalas de objetos, algunos métodos sugieren procesar la imagen utilizando varios tamaños de ésta y combinar los resultados después. Sin embargo, en esta técnica, al utilizar capas de características de varias capas diferentes en una sola red neuronal para la predicción, se puede imitar el mismo efecto.

Basándose en otros estudios, los creadores de esta técnica decidieron utilizar capas de características tanto en las capas inferiores como las superiores de la red. La Figura 4.1 muestra dos ejemplos de mapa de características, 4x4 y 8x8, que se utilizan en el modelo. En la práctica se pueden utilizar mucho más con una pequeña sobrecarga computacional.

Dentro del marco de SSD se diseña el mosaico de las cajas por defecto para que los mapas de características específicas aprendan a responder a diferentes escalas de

4.1. TÉCNICA SINGLE SHOT MULTIBOX DETECTION

objetos. En la práctica se puede diseñar una distribución de cuadros predeterminados para que se ajuste mejor a un conjunto de datos específico.

Combinando predicciones para todas las cajas por defecto con diferentes escalas y relaciones de aspecto de todas las ubicaciones de los mapas de características se consigue un conjunto diverso de predicciones, abarcando varios tamaños y formas de objetos de entrada. Por ejemplo en la Figura 4.1, el perro está emparejado con una caja por defecto en el mapa de características 4×4 , pero con ninguna en el mapa de características 8×8 . Esto se debe a que esas cajas tienen escalas diferentes y no coinciden con la caja del perro, y por lo tanto se consideran como negativos durante el entrenamiento.

- **Ejemplos 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 significativo 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.

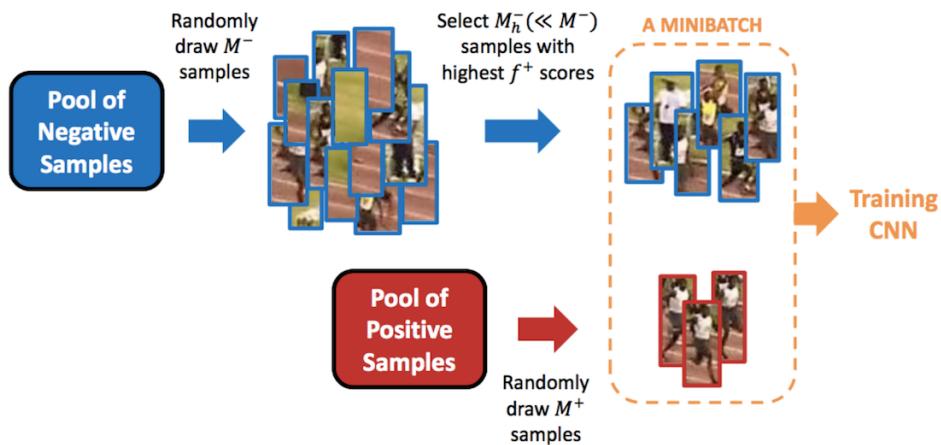


Figura 4.4: Ejemplo de uso de negativos en el entrenamiento SSD [30].

- **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.

4.2. Aplicación para detección de objetos en flujos de vídeo con SSD

Con el objetivo de realizar la detección neuronal en vivo se ha programado un componente en Python que, a partir del *Camera Server* de JdeRobot explicado en el apartado 3.2.1, será capaz de realizar la detección en tiempo real de las imágenes captadas por la cámara. De esta manera se abre el camino para incorporar esta capacidad a la parte perceptiva de un robot móvil o a un sistema de videovigilancia, por ejemplo.

4.2.1. Diseño

El cómputo dentro de la aplicación se ha descompuesto en 3 hilos de ejecución concurrentes. La Figura 4.5 muestra un diagrama de bloques con el diseño de este componente.

- *Hilo Camera*: Será el encargado de capturar las imágenes captadas por la cámara utilizando para ello el servidor de imágenes *Camera Server*. Esta imagen será enviada al *hilo GUI* para su visualización y al *hilo Detector* para su procesamiento.
- *Hilo Detector*: Este hilo será el encargado de la detección neuronal usando redes SSD. Para ello hace las transformaciones que sean necesarias de la imagen que recibe desde el *hilo Camera* y realiza la detección de ésta. Tras generar como salida la misma imagen con las *bounding boxes* de los objetos detectados superpuestas, la envía al *hilo GUI* para su visualización.
- *Hilo GUI*: Es la interfaz gráfica de usuario, o Graphical User Interface, GUI, es decir, el módulo encargado del aspecto gráfico de la aplicación, mostrando tanto

la imagen real captada por la cámara como la imagen tras su procesamiento y la detección de objetos.

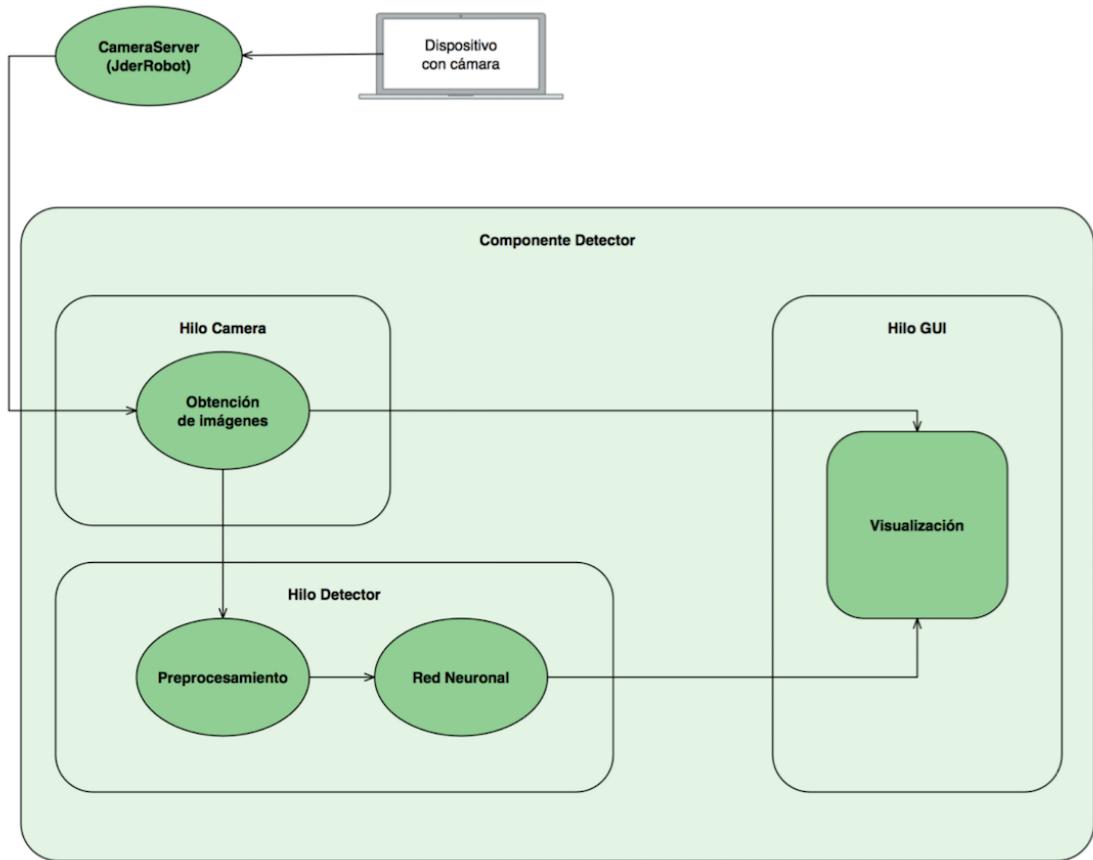


Figura 4.5: Arquitectura de la aplicación de detección visual con SSD.

4.2.2. Hilo Camera

Este hilo es el encargado de obtener la imagen capturada por la cámara, la cual es visualizada en el hilo encargado de la parte gráfica. Este hilo puede ser referenciado con el nombre de la clase, *Camera()*.

Al inicializar esta clase, en el constructor, se especifica que se va a realizar una referencia externa, en forma de fichero de configuración. De esta manera, el *hilo Camera* se puede conectar con el componente JdeRobot *Camera Server*, y así adquirir la imágenes que éste provee. El fichero de configuración, que se especifica en línea de comandos cuando

se ejecuta el componente, se muestra a continuación.

```
DetectorSSD.Camera.Proxy=cameraA:default -h localhost -p 9999
```

Tras el constructor de la clase y una vez que la aplicación está conectada con el componente JdeRobot, se definen dos funciones de gran importancia para la ejecución de ésta. En primer lugar, el método *getImage(self)* cuya función será obtener las imágenes de la cámara. También realiza una serie de transformaciones de éstas antes de que el *hilo Detector* las introduzca en la red neuronal. Segundo, la función *update(self)* cuyo objetivo es actualizar la ejecución de este hilo. Este método es llamado desde la clase *ThreadCamera* en la que se define que el ciclo de ejecución de este hilo es de 1500 milisegundos.

4.2.3. Hilo Detector

Este hilo es el encargado de todo el proceso de detección neuronal, desde el procesamiento de la imagen que recibe desde el *hilo Camera* para adaptarla a lo que requiere la red neuronal, hasta la obtención de la salida de la red y la posterior generación de la imagen con las detecciones producidas.

Para realizar la detección, este hilo utiliza el mismo proceso explicado en el apartado 4.3, en el cual, a partir de una única imagen de entrada se genera como salida la misma imagen con las *bounding boxes* de los objetos detectados superpuestos sobre ella. Así, en el constructor de este hilo se definen las variables que hacen referencia al mapa de etiquetas que pueden ser asignadas en la detección, a la estructura de la red y a los pesos de la red entrenada. Tras esto, se define la función *detectiontest(self,img)*, explicada en el apartado 4.3 y que se encarga de introducir la imagen de entrada en la red neuronal y obtener las detecciones.

Finalmente, como en el *hilo Camera*, se define la función *update(self)* que actualizará la ejecución de este hilo. El ciclo de ejecución está especificado en la clase *ThreadDetector* y es de 1500 milisegundos.

4.2.4. Hilo GUI

Este hilo será el encargado de todo el aspecto gráfico de la aplicación. Renderiza las dos imágenes generadas en el resto de hilos, es decir, la capturada en tiempo real por el *hilo Camera* y la imagen procesada en la que se muestra los objetos detectados recibida desde el *hilo Detector*.

En el constructor de esta clase, en primer lugar se inicializa la ventana principal, en la que aparece todos los elementos comentados anteriormente.

```
QtGui.QWidget.__init__(self, parent)
self.setWindowTitle("Detection Component")
self.resize(1200,500)
self.move(150,50)
self.updGUI.connect(self.update)
```

En el código anterior se define el título que aparecerá en la parte superior de la ventana, *setWindowTitle*, la anchura y altura de ésta, *resize* y su posición, *move*. Con la función *connect* se le asocia la función *update*, que se encargará de actualizar este hilo de ejecución.

Posteriormente, se definen las ventanas donde se visualizarán las dos imágenes. Éstas vienen parametrizadas igualmente por los parámetros *resize* y *move*.

Finalmente, se crean dos botones encargados de controlar el proceso de detección. Estos botones, además de venir definidos por del parámetro *move* para determinar su posición, se les añadirá la función *setStyleSheet* con el objetivo de asociarles un color. Adicionalmente, utilizando *clicked.connect* se les asignará la función que se ejecutará cuando se haga *click* sobre cada uno de ellos.

Tras el constructor de la clase GUI, se define la función *update(self)* que actualiza la ejecución de este hilo. Aquí, se asigna a cada una de las ventanas creadas la imagen captada de la cámara y la imagen con las detecciones obtenidas según corresponda.

Finalmente, se definen las dos funciones asociadas a los dos botones creados anteriormente. La función *toggle(self)* estará asociada al botón *Continuous* y hará que tras

pulsarlo, comience la detección continua de las imágenes captadas en tiempo real. Por otro lado, la función *detectOnce(self)* estará asociada al botón *Detect Once* y hará que al hacer *click* sobre él, se realice la detección únicamente sobre el frame que se estaba mostrando por la cámara en ese momento exacto.

4.2.5. Ejecución

El proceso de ejecución de este esta aplicación está dividida en dos pasos. Uno es el encargado de lanzar el servidor de imágenes, utilizando para ello el componente *Camera Server* de JdeRobot. El otro es el encargado de lanzar el propio aplicación de detección.

Para lanzar el componente *Camera Server*, será necesario abrir un terminal y ejecutar el siguiente comando proporcionado por la plataforma JdeRobot.

```
cameraserver cameraserver.cfg
```

Una vez que está lanzado el servidor de imágenes, se ejecutará el siguiente comando, el cual lanzará la propia aplicación detectora.

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

Se especifica el fichero de configuración *detectorSSD.cfg* gracias al cual se tendrá comunicación con el servidor de imágenes. Este fichero tiene una propiedad en la que se define el nombre de la cámara, el cual tiene que ser el mismo que el definido anteriormente en el fichero *cameraserver.cfg* para que dicha conexión sea efectiva.

En la Figura 4.6 se puede observar cómo se comporta la aplicación, en este caso, pulsando el botón encargado de realizar la detección única.

4.2. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

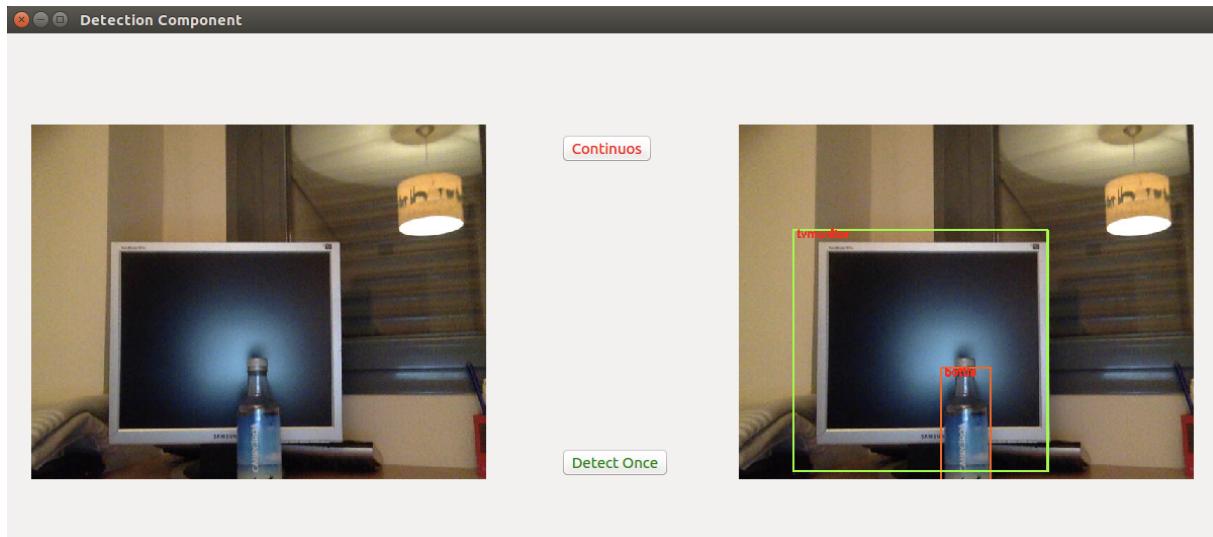


Figura 4.6: Ejecución de la aplicación usando el botón de detección única.

4.2.6. Detección específica de un tipo de objetos

En ocasiones, se quiere realizar tareas de detección sobre uno o varios objetos en concreto, obviando los demás que se han definido durante el entrenamiento de la red neuronal. Para ello, en el fichero de configuración, se añadirá una línea en la cual se especifican las etiquetas de los objetos que se quieren detectar.

Por ejemplo se modifica este fichero en función de la demostración realizada anteriormente, en la que se detectaron una pantalla y una botella.

```
DetectorSSD.Camera.Proxy=cameraA:default -h localhost -p 9999  
#Write all objects that yo want to detect separated by one space.  
DetectorSSD.Labels=bottle
```

Tras esto, a partir de la variable `DetectorSSD.Labels` creada, se recorrerá el fichero `labelmap.prototxt`, el cual contiene todos los objetos para los que se ha preparada a la red. Si alguna de las etiquetas existentes coinciden con las especificadas en el fichero de configuración, se creará el archivo `labelmap-conditional.prototxt`, que contendrá los objetos definidos. De esta manera, se tendrá un mapa de etiquetas con únicamente los objetos sobre los que se quiere realizar la detección.

En la Figura 4.7 se muestra la salida de este componente de detección específica, utilizando el fichero de configuración mostrado anteriormente:

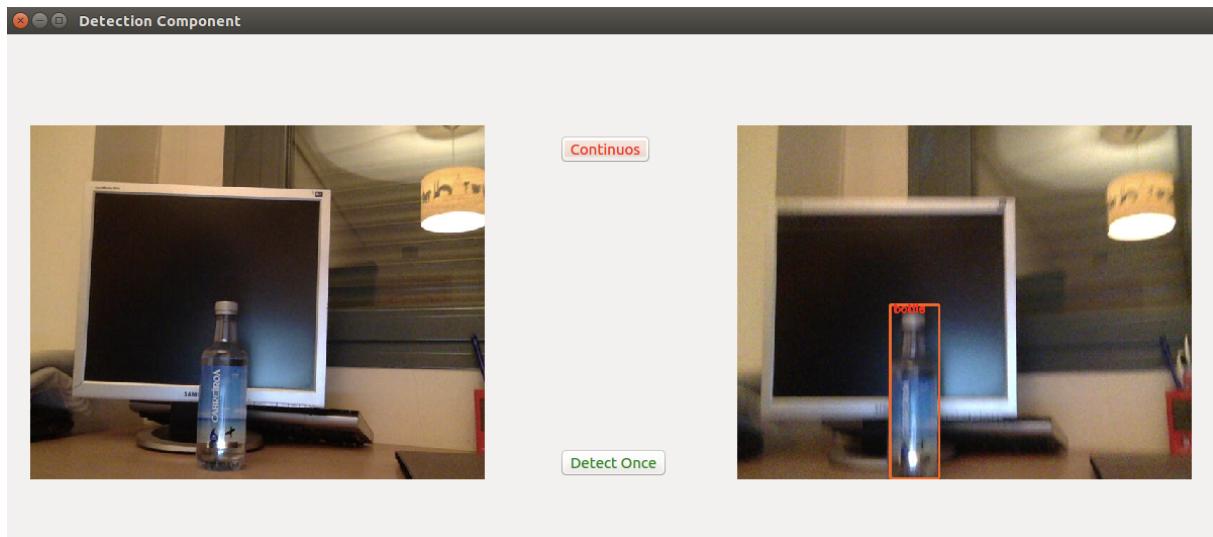


Figura 4.7: Ejecución del componente detector condicional usando el botón de detección única.

4.3. Uso de redes SSD preentrenadas en Caffe

Tras entender el funcionamiento de la técnica SSD, se realizará un ejemplo práctico que proporciona la herramienta Caffe. En este primer caso, se utilizarán dos de los modelos preentrenados proporcionados en el repositorio oficial de los desarrolladores de la técnica SSD¹. Particularmente uno que utiliza la base de datos PascalVOC y otro que utiliza COCO.

- **PascalVOC07+12-SSD300x300:** Se trata de un modelo entrenado por el conjunto de las imágenes de entrenamiento de la base de datos PascalVOC de 2007 y 2012, con unas dimensiones de 300x300 píxeles.
- **COCO-SSD300x300:** Se trata de un modelo entrenado utilizando la base de datos COCO. Se utiliza una dimensión de imagen de 300x300 píxeles.

Tras definir los modelos que se usarán, se crea el script *image-detection.py*², el cual, a

¹<https://github.com/weiliu89/caffe/tree/ssd>

²https://github.com/RoboticsURJC-students/2016-tfg-David-Butragueno/blob/master/image_detection.py

4.3. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

partir de una imagen de entrada, realizará una serie de transformaciones sobre esta con el objetivo de adaptarla a la entrada de la red neuronal, la introducirá dentro de ésta y finalmente generará como salida esa misma imagen con las *bounding boxes* sobre los objetos detectados. Este script será creado a partir del fichero *ssd_detect.ipynb*³, desarrollado por los creadores de SSD.

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

- **labelmap.prototxt:** Define las etiquetas que pueden ser asignadas en la detección.
- **model_def:** Define la estructura básica de la red neuronal, la cual será explicada en el apartado 4.4.1.
- **model_weights:** Se trata de un fichero binario que contiene el estado actual de los pesos para cada capa de la red neuronal.

Tras esto, utilizando la función de Caffe *Net*, se define la red neuronal al completo, indicando tanto su estructura como los pesos de cada una de las capas, ambos datos almacenados en las variables *model_def* y *model_weights* comentadas anteriormente.

```
self.net = caffe.Net(model_def, # defines the structure of the model  
                      model_weights, # contains the trained weights  
                      caffe.TEST)    # use test mode
```

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
```

³https://github.com/weiliu89/caffe/blob/ssd/examples/ssd_detect.ipynb

```

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 descompuesta 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 deseé. Para ello, se usa

4.4. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

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]
```

A continuación se muestran dos ejemplos de la salida de este *script* utilizando los modelos comentados anteriormente y una imagen del conjunto de test de la base de datos PascalVOC, la cuál será utilizada para realizar las pruebas explicadas en el apartado 5.3



Figura 4.8: (a) Detección con modelo entrenado con PascalVOC. (b) Detección con modelo entrenado con COCO.

Como se puede observar en la Figura 5.1, el modelo entrenado con COCO ha detectado una persona, un coche y una moto mientras que el modelo entrenado con PascalVOC únicamente ha detectado el coche y la moto.

4.4. Creación y uso de un modelo de red SSD propio

Para conseguir el sistema detector de objetos que se pretende, antes hay que definir en varios aspectos que afectarán a las prestaciones que tendrá el modelo una vez construido. En esta sección se explicará la arquitectura que tendrá la red neuronal que se utilizará, los parámetros que se usarán para el entrenamiento de ésta y los conjuntos de datos de entrenamiento con los que se alimentará a la red.

4.4.1. Estructura de la red

La plataforma Caffe define la estructura de las redes neuronales en ficheros con extensión *prototxt* [29]. Aquí se definen todas las capas existentes en el modelo además de las características propias de cada una de ellas. En este caso en particular, la estructura de la red neuronal propia tiene un arquitectura similar a la definida en la técnica SSD, explicada en el apartado 4.1.1. En primer lugar aparecen varias redes de convolución y *pooling*, copiando la estructura de las redes convolucionales, con el objetivo de disminuir el tamaño de los ejemplos utilizados para el entrenamiento y poder capturar las características de éstos. Finalmente, se añaden varias capas encargadas de transformar las imágenes de entrada para detectar los objetos existentes en ellas. A continuación se describe la funcionalidad de cada una de las capas existentes en el modelo.

En primer lugar, se especifica el nombre de la red, en este caso 'VGG_VOC0712_COCO_SSD_300x300'.

```
name: "VGG_VOC0712_COCO_SSD_300x300"
```

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 propia de la siguiente manera:

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

4.4. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

```
    dim: 300  
}
```

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 3.1.3.2.

```
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 3.1.3.2.

```
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 continuos 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 de convolución y pooling, se intercalan varias capas de activación, utilizando la función ReLU, explicada en el apartado 3.1.3.5.

```
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 la 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
}
}
}
```

Los números positivos se utilizan directamente, configurando la dimensión correspondiente en la capa de salida. Además, se aceptan dos 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 dos 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 i .

4.4. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

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

Finalmente, se define la capa que determina la salida, que determinará los parámetros necesarios para clasificar los objetos detectados durante el entrenamiento.

```
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: 0.45
            top_k: 400
        }
        save_output_param {
            label_map_file: "data/VOC0712/labelmap_voc.prototxt"
        }
        code_type: CENTER_SIZE
        keep_top_k: 200
        confidence_threshold: 0.01
    }
}
```

Además de especificar las capas alimentarán la entrada de ésta, se pueden destacar

los parámetros *num_classes*, que determina el número de clases de objetos existentes o *label_map_file* que indica la ruta del fichero dónde se definen estas clases.

4.4.2. Definición del solucionador

El solucionador es el responsable de la optimización del modelo y se define en un archivo con extensión *.prototxt*. Aquí, se especifican los parámetros necesarios para ejecutar de forma correcta el entrenamiento. Más concretamente, el solucionador se encarga de:

- Crea la red de entrenamiento para el aprendizaje y la red de test para la evaluación.
- Optimiza de forma iterativa realizando llamadas hacia delante y hacia atrás de la red, actualizando al mismo tiempo los parámetros de esta.
- Periódicamente, evalúa las redes de prueba.
- Crea instantáneas del modelo y del estado del solucionador a lo largo de la optimización.

En cada iteración, el solucionador realiza las siguientes operaciones:

- Cálculo de la salida de la red y de la pérdida.
- Llamada hacia atrás a la red para calcular el gradiente.
- Incorpora los gradientes en las actualizaciones de los parámetros según el método del solucionador.
- Actualiza el estado del solucionador de acuerdo con la velocidad de aprendizaje, el historial y el método.

A continuación, se muestra el aspecto del solucionador usado para el entrenamiento de la red, llamado *solver.prototxt*:

```
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: 0.001
```

```

display: 10
max_iter: 120000
lr_policy: "multistep"
gamma: 0.1
momentum: 0.9
weight_decay: 0.0005
snapshot: 80000
snapshot_prefix: "models/VGGNet/VOC0712/SSD_300x300/VGG_VOC0712_SSD_300x300"
solver_mode: GPU
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"

```

Uno de los primeros parámetros que hay que especificar es *type*, el cual determina el tipo de solucionador que se va a utilizar. En Caffe existen 6 tipos diferentes de solucionadores, cuyas características y funcionalidades se pueden consultar en la documentación oficial de Caffe⁴. Para este caso en particular, se ha utilizado el tipo SGD, *Stochastic Gradient Descent*.

Otros de los parámetros que se pueden destacar dentro este fichero son, *train_net* y *test_net* que muestran la localización de los ficheros que especifican la estructura de la red; *solver_mode*, que especifica si se va a utilizar la unidad central o la unidad gráfica de procesamiento, en este caso será CPU; *base_lr*, que fija el ritmo de aprendizaje, *learning rate*, durante el entrenamiento; *lr_policy* que determina la política que seguirá la ritmo de aprendizaje, en este caso será *step*, que indica que cada cierto tiempo se reducirá la

⁴<http://caffe.berkeleyvision.org/tutorial/solver.html>

tasa de aprendizaje; *gamma* que especifica cuanto se va a reducir el *learning rate* cada *step*; *stepsize* que determina cada cuantas iteraciones o *steps* se reducirá el ritmo de aprendizaje; *max_iter*, que define el número máximo de iteraciones que se van a realizar en el entrenamiento;

4.4.3. Entrenamiento de la red

Antes de comenzar el entrenamiento de la red neuronal, es necesario definir las imágenes con las que vamos a alimentar a ésta durante el proceso. Para este caso, se decidió crear una base de datos combinada la cuál contiene tanto las imágenes de entrenamiento de PascalVOC de los años 2007 y 2012 como las de COCO. Para componerla fue necesario obtener los ficheros *.lmdb* que incluyen toda la información de ambas bases de datos. Para ello, se dispone de los siguientes recursos y documentación:

- En el repositorio Git de los desarrolladores de la técnica SSD⁵, en el apartado **Preparation**, se explica paso a paso todo el proceso para crear los ficheros LMDB que contendrán las imágenes y anotaciones de la base de datos PascalVOC.
- En el repositorio Git referenciado⁶ se explica como crear los ficheros LMDB que contendrán la información de la base de datos COCO.

Tras esto, se creó el *script* *create_lmdb.py*⁷ que será el encargado de fusionar en un único fichero LMDB toda la información de ambas bases de datos existente en los archivos creados anteriormente.

Para el entrenamiento de la red neuronal, los desarrolladores de SSD generaron varios *scripts* que cumplen este objetivo. Particularmente, se destacan los *scripts* *ssd_pascal.py* y *ssd_coco.py*, facilitados en el repositorio oficial de los creadores de esta técnica, y cuya finalidad es el entrenamiento de la red utilizando las imágenes de entrenamiento y validación de las bases de datos PascalVOC y COCO respectivamente. En estos *scripts* se encuentran tanto las rutas de todos los ficheros necesarios para el entrenamiento, ya sea el modelo, el mapa de etiquetas, el solucionador o los archivos que alimentaran a la red

⁵<https://github.com/weiliu89/caffe/tree/ssd>

⁶<https://github.com/intel/caffe/tree/master/data/coco>

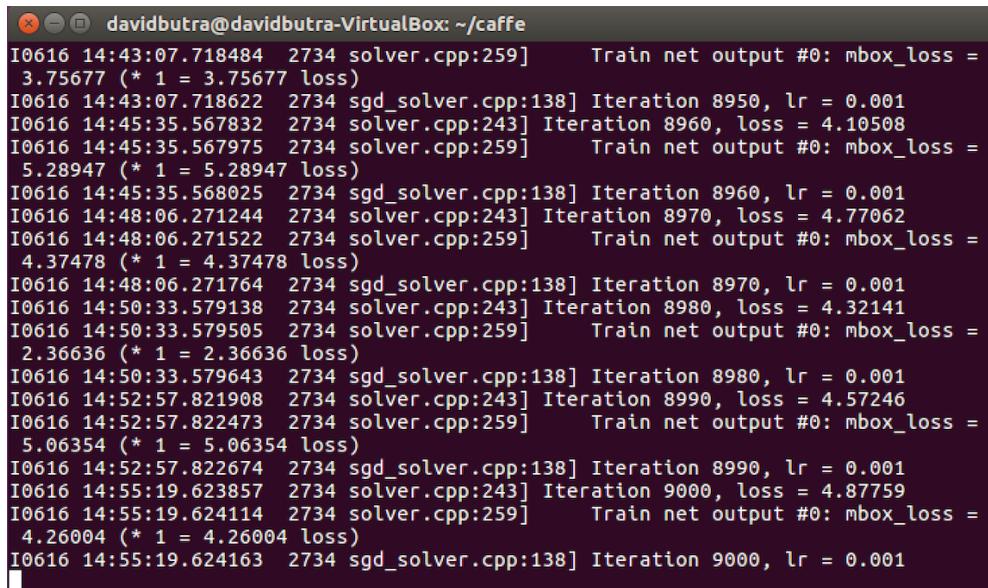
⁷https://github.com/RoboticsURJC-students/2016-tfg-David-Butragueno/blob/master/create_lmdb.py

4.4. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

de imágenes durante el proceso, como las rutas dónde se almacenaran los resultados del entrenamiento. Finalmente, a partir de los datos provistos, se crean la red neuronal y el solucionador, necesarios para lanzar el entrenamiento.

Para este caso se creó un *script* adicional, a partir de los mencionados anteriormente, modificando todos los datos necesarios. Tras la ejecución de este *script*, comienza el entrenamiento, mostrando por la pantalla del terminal información del propio proceso cada 10 iteraciones.

Durante la ejecución, cada 1000 iteraciones, se genera una *snapshot* en forma de fichero binario con extensión *.caffemodel* que contiene los pesos existentes en cada una de las capas de la red neuronal en ese momento del entrenamiento.



The image shows a terminal window titled "davidbutra@davidbutra-VirtualBox: ~/caffe". The window displays a log of training progress. The log consists of several lines of text, each starting with "I0616" followed by a timestamp, a process ID (2734), and a file name like "solver.cpp". The log entries indicate the training of a neural network, specifically monitoring the "mbox_loss" metric. The loss values fluctuate over time, with some lines showing intermediate snapshots and others showing the final iteration at 9000. The log ends with the final iteration at 9000 and a learning rate of 0.001.

```
I0616 14:43:07.718484 2734 solver.cpp:259]      Train net output #0: mbox_loss = 3.75677 (* 1 = 3.75677 loss)
I0616 14:43:07.718622 2734 sgd_solver.cpp:138] Iteration 8950, lr = 0.001
I0616 14:45:35.567832 2734 solver.cpp:243] Iteration 8960, loss = 4.10508
I0616 14:45:35.567975 2734 solver.cpp:259]      Train net output #0: mbox_loss = 5.28947 (* 1 = 5.28947 loss)
I0616 14:45:35.568025 2734 sgd_solver.cpp:138] Iteration 8960, lr = 0.001
I0616 14:48:06.271244 2734 solver.cpp:243] Iteration 8970, loss = 4.77062
I0616 14:48:06.271522 2734 solver.cpp:259]      Train net output #0: mbox_loss = 4.37478 (* 1 = 4.37478 loss)
I0616 14:48:06.271764 2734 sgd_solver.cpp:138] Iteration 8970, lr = 0.001
I0616 14:50:33.579138 2734 solver.cpp:243] Iteration 8980, loss = 4.32141
I0616 14:50:33.579505 2734 solver.cpp:259]      Train net output #0: mbox_loss = 2.36636 (* 1 = 2.36636 loss)
I0616 14:50:33.579643 2734 sgd_solver.cpp:138] Iteration 8980, lr = 0.001
I0616 14:52:57.821908 2734 solver.cpp:243] Iteration 8990, loss = 4.57246
I0616 14:52:57.822473 2734 solver.cpp:259]      Train net output #0: mbox_loss = 5.06354 (* 1 = 5.06354 loss)
I0616 14:52:57.822674 2734 sgd_solver.cpp:138] Iteration 8990, lr = 0.001
I0616 14:55:19.623857 2734 solver.cpp:243] Iteration 9000, loss = 4.87759
I0616 14:55:19.624114 2734 solver.cpp:259]      Train net output #0: mbox_loss = 4.26004 (* 1 = 4.26004 loss)
I0616 14:55:19.624163 2734 sgd_solver.cpp:138] Iteration 9000, lr = 0.001
```

Figura 4.9: Ejecución del entrenamiento de la red neuronal.

Por último, cuando finalizan las iteraciones definidas en el solucionador, en este caso 120000, se crea el fichero final con los pesos de la red una vez terminado el entrenamiento, con el mismo formato que el de las *snapshots* comentadas.

Capítulo 5

Medidor de calidad

5.1. Comparación *bounding boxes* real y detectada

En el apartado 4.3 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 3.3, 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 dos 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.

5.1.1. 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/d-b/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 si-

5.1. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

guiente 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 dos *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 dos 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.

5.1. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

```
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 5.1 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 5.1: Comparacion bounding boxes real y detectada con imagen de la BBDD PascalVOC.

5.1.2. COCO

Como se ha explicado en la sección 3.3.2, 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 de 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 3.7, 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

5.1. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

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 dos 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 dos 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 coordenadas 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. La imagen 5.2 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.

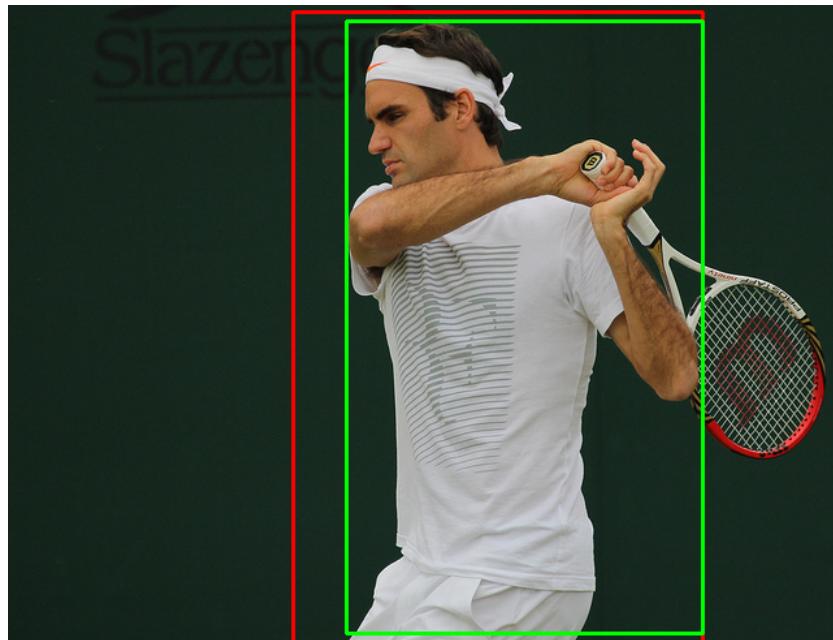


Figura 5.2: Comparación bounding boxes real y detectada con imagen de la BBDD COCO.

5.2. Métricas de evaluación

Existen varios parámetros cuya cálculo e interpretación hacen posible la evaluación de la capacidad que tienen determinados sistemas a la hora de detectar objetos. En esta sección se explicarán los que se calcularán posteriormente en el apartado 5.3 tras evaluar

con *Detection Suite* las redes utilizadas durante este trabajo.

5.2.1. Precisión y Recall

Varias de las métricas utilizadas a la hora de evaluar el rendimiento de modelos creados para la detección de objetos se enfocan en la capacidad que tiene el detector para realizar buenas detecciones. Para este objetivo, se pueden destacar dos medidas:

- **Precisión**, mide cómo de precisas son las detecciones, es decir, qué porcentaje de las predicciones son correctas.

$$Precision = \frac{TP}{TP + FP} \quad (5.1)$$

donde TP es el número total de verdaderos positivos y FP el número de falsos positivos.

- **Recall**, mide cómo de bien el modelo encuentra los positivos.

$$Precision = \frac{TP}{TP + FN} \quad (5.2)$$

donde FP es el número de falsos negativos.

Utilizando estas métricas, un determinado modelo de detección, además de devolver las detecciones realizadas, devolverá las probabilidades de que dichas detecciones sean acertadas. Fijando diferentes umbrales, se pueden obtener los valores *Average Precision* y *Average Recall* que hacen referencia a la precisión y *recall* media para un determinado umbral y para las diferentes clases existentes en un modelo. Normalmente, los detectores de objetos suelen tener múltiples clases y es posible que se necesita medir el promedio de precisión o *recall* medio para todas las clases. Así, los parámetros *mean Average Precision* y *mean Average Recall* dan una visión más general de cómo el detector es capaz de actuar para todas las clases. Por ejemplo, el mAP para N clases:

$$mAP = \frac{1}{N} \sum_{class=1}^N AP_{class} \quad (5.3)$$

5.2.2. 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

5.2. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

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.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \quad (5.4)$$

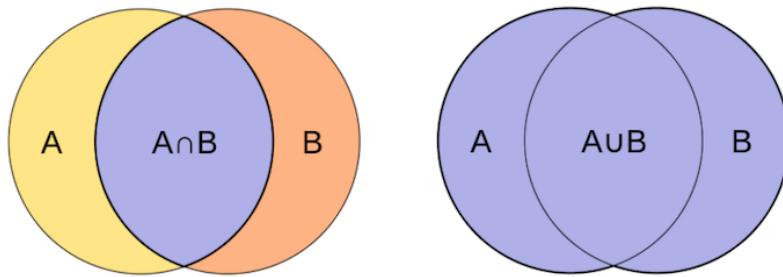


Figura 5.3: Intersección y unión sobre dos conjuntos A y B.

Si no hay coincidencias en ambos conjuntos, el índice Jaccard se definirá de la forma $J(A, B) = 1$. Para cualquier coincidencia de los dos conjuntos, el IoU estará comprendido entre 0 y 1, tal y como se define en la ecuación 5.5.

$$0 \leq J(A, B) \leq 1 \quad (5.5)$$

Por lo tanto, el índice Jaccard o IoU puede definir gráficamente tal y como representa la imagen 5.4:

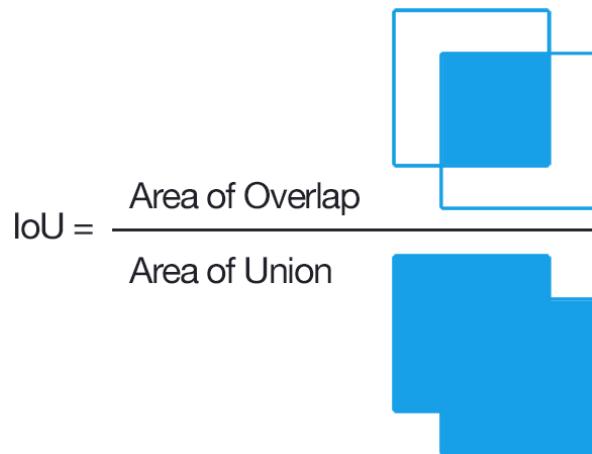


Figura 5.4: Intersection over Union [33].

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 5.5: Evaluación IoU [33].

5.3. Resultado final

En este apartado se procederá a realizar las pruebas necesarias para evaluar la calidad con la que detectan objetos los modelos comentados durante la memoria. Las redes entrenadas que se medirán, serán las 2 preentrenadas con las bases de datos PascalVOC y COCO, enunciadas en el apartado 4.3, y la entrenada utilizando una base de datos personalizada explicada en el apartado 4.4.

Para medir las prestaciones de cada uno de los modelos, se va a utilizar la herramienta *Detection Suite*¹ de JdeRobot, comentada en el apartado 1.3. Esta herramienta permite

¹<https://github.com/JdeRobot/dl-DetectionSuite>

5.3. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

la utilización de varias bases de datos, entre las que se encuentran las utilizadas durante este trabajo, PascalVOC y COCO, además de varios frameworks para el aprendizaje profundo, incluyendo Caffe.

Como datos de entrada para medir el rendimiento de los modelos, con *Detection Suite* se pueden utilizar un banco de imágenes con sus respectivas anotaciones, vídeos grabados o las imágenes que capta una cámara en tiempo real. Para este caso, se decidió utilizar un *testing set* formado por imágenes junto con sus anotaciones. Lo más importante, es que estas imágenes seleccionadas no hayan sido utilizadas para entrenar ninguno de los modelos que se compararán, ya que esto falsearía las medidas obtenidas. En el repositorio de la técnica SSD², en el apartado **Preparation**, hay un enlace para descargarse varias imágenes junto con sus anotaciones del conjunto de test de la base de datos PascalVOC. Este banco de ejemplos está formado por 4952 imágenes en los que hay etiquetados 14976 objetos de 20 clases diferentes. En la figura 5.6 se puede observar como están distribuidos los 14976 objetos mencionados anteriormente en cada una de las clases existentes:

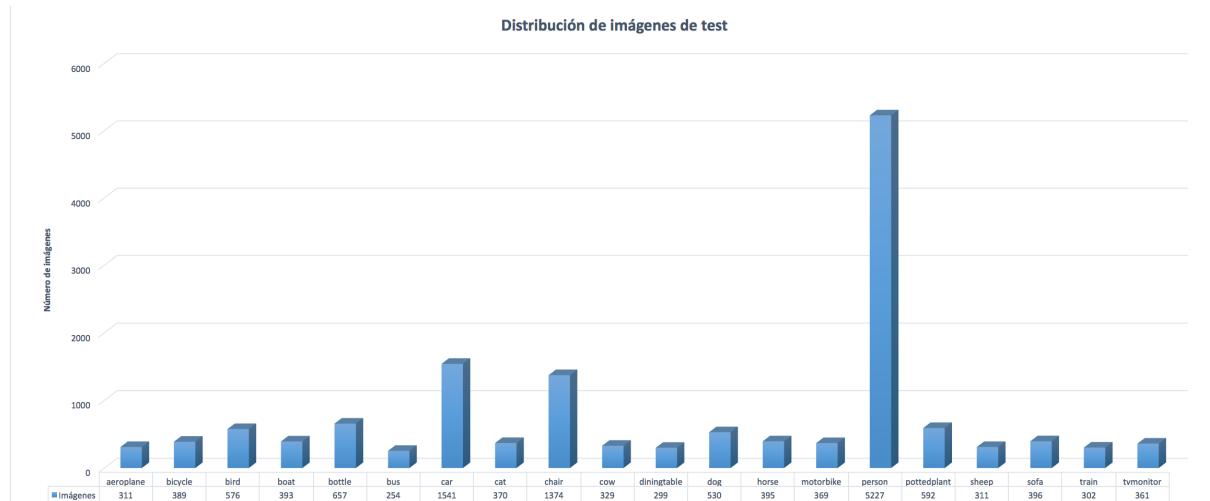


Figura 5.6: Distribución de las imágenes de test.

Dentro de *Detection Suite* existen varias herramientas para realizar mediciones de modelos entrenados. En este caso, se usarán 2 de ellas existentes en su interfaz gráfica. En primer lugar, se utilizará la herramienta *Detector*³ que ejecuta modelos entrenados sobre un conjunto de datos de prueba, generando así la detección de objetos sobre las imágenes

²<https://github.com/weiliu89/caffe/tree/ssd>

³<https://github.com/JdeRobot/DetectionSuite/wiki/Detector>

que lo forman. La imagen 5.7 muestra el aspecto de esta interfaz.

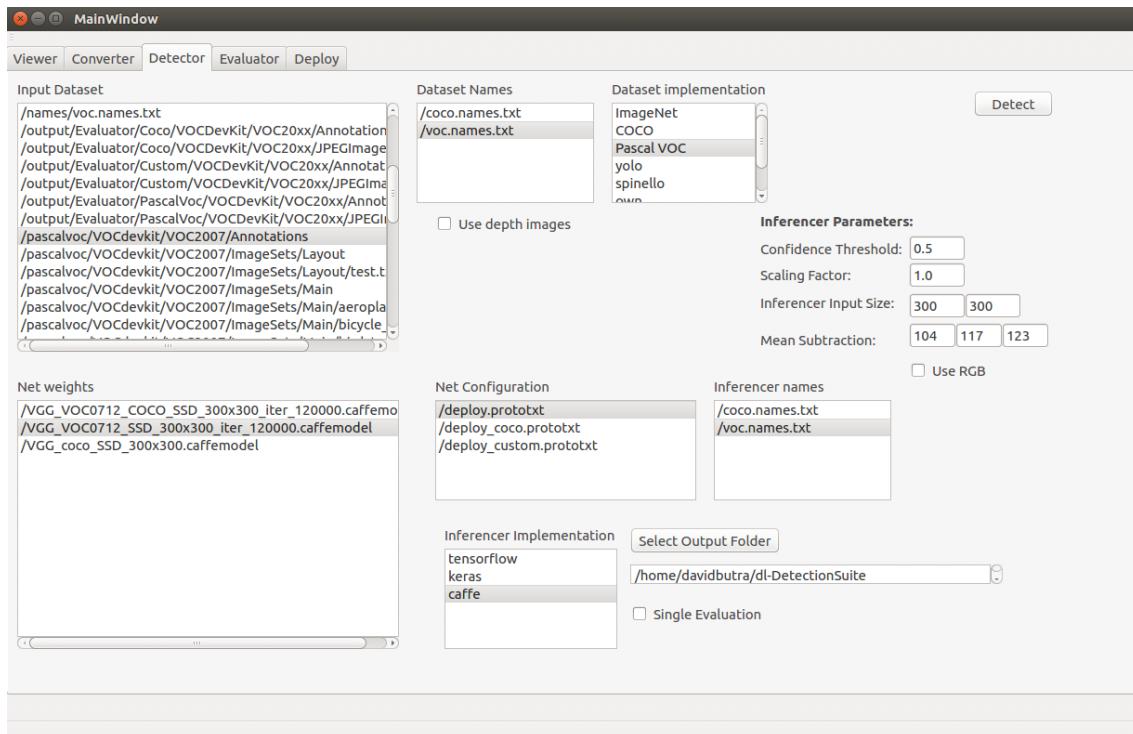


Figura 5.7: Interfaz gráfica de la herramienta *Detector*.

Como se muestra en la imagen anterior, para utilizar esta herramienta hay que definir varios parámetros como *Input Dataset*, dónde se establece la ruta de los ficheros de anotaciones del conjunto de datos, así como de las imágenes, las cuales deben estar en el mismo directorio, *Net Weights*, dónde se definen los pesos de la red que se quiere evaluar, *Dataset Names* que hace referencia a las etiquetas de los objetos existentes, *Net Configuration* que especifica la estructura de la red neuronal. En el apartado *Inferencer Implementation*, se define el framework utilizado dentro de los disponibles en la aplicación⁴. Para el caso exclusivo de Caffe, hay que especificar varios parámetros adicionales tales como el umbral de confianza en las detecciones realizadas, definido para estos casos en 0.5 y el factor de escala, dimensiones de las imágenes de entrada y sustracción media para los canales de color BGR, obtenidos del repositorio de *Detection Suite*⁵ en función del modelo utilizado. Finalmente, es posible especificar la carpeta en la que se desea que

⁴<https://github.com/JdeRobot/DetectionSuite/wiki/FrameWorks>

⁵<https://github.com/JdeRobot/DetectionSuite/wiki/Model-Zoo>

5.3. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

se guarden los resultados obtenidos.

Tras definir todos los parámetros, es posible comenzar con la detección pulsando el botón *Detect*. Mientras se está ejecutando esta herramienta, se van mostrando todas las imágenes con las *bounding boxes* de los objetos detectados y por terminal la confianza de cada detección, como se muestra en la imagen 5.8.

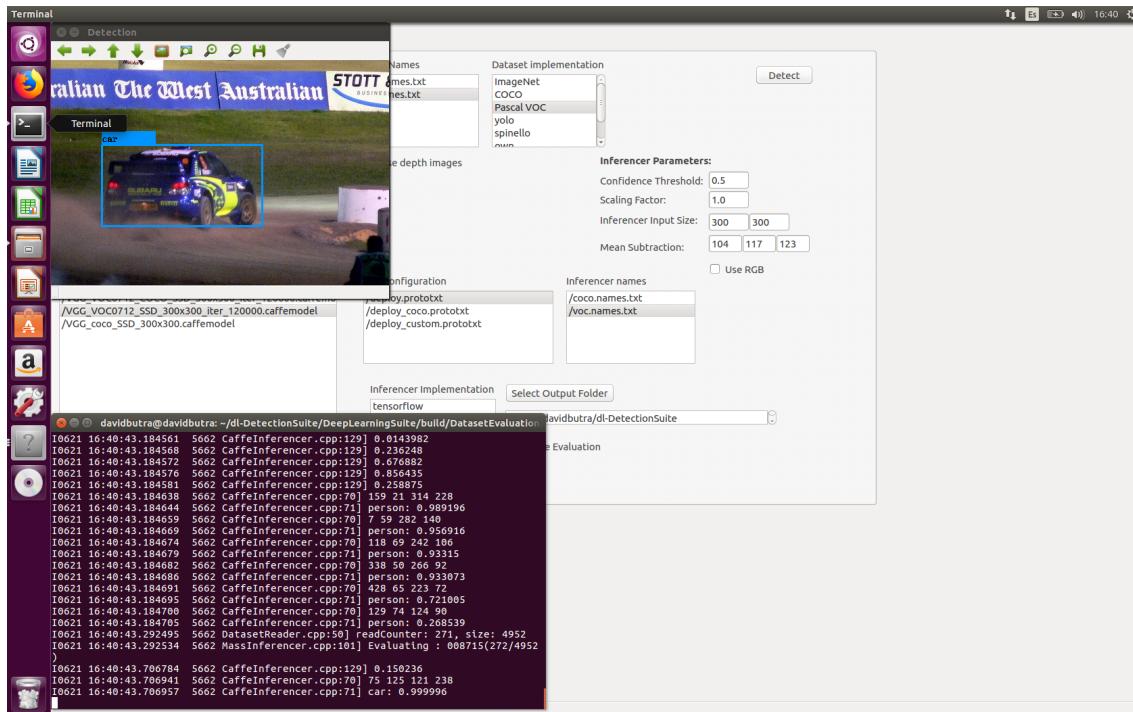


Figura 5.8: Ejecución de la herramienta *Detector*.

Cuando termina la ejecución, se genera la salida en forma de fichero de anotaciones. Como se puede observar en la imagen 5.7 se ha especificado como implementación de base de datos, *Dataset Implementation*, PascalVOC, por lo que las anotaciones generadas tendrán las características de las de esta base de datos, explicadas en el apartado 3.3.1. Por lo tanto, se crearán tantos ficheros XML como imágenes, es decir, 4952, dónde se especificarán las coordenadas de cada uno de los objetos detectados.

Tras la utilización del detector, se usará la herramienta *Evaluator*⁶ de *Detection Suite*. Esta herramienta evalúa los resultados obtenidos anteriormente con el detector generando

⁶<https://github.com/JdeRobot/DetectionSuite/wiki/Evaluator>

un archivo CSV que contiene los resultados generales además de los específicos para cada una de las clases existentes en el conjunto de datos dado. Las métricas calculadas son el mAP y mAR conjunto para todas las clases y para cada una de las clases utilizando 10 umbrales de IoU entre 0.5 y 9.5, además del cómputo de tiempo de evaluación. Los resultados obtenidos por *Detection Suite* están basados en la API de COCO⁷, por lo que, como se dice en su página oficial⁸, no habrá distinción entre *Average Precision* y mAP, al igual que entre *Average Recall* y mAR, ya que se asume que la diferencia está clara en el contexto. La imagen 5.9 muestra el aspecto de la interfaz del evaluador de *Detection Suite*.

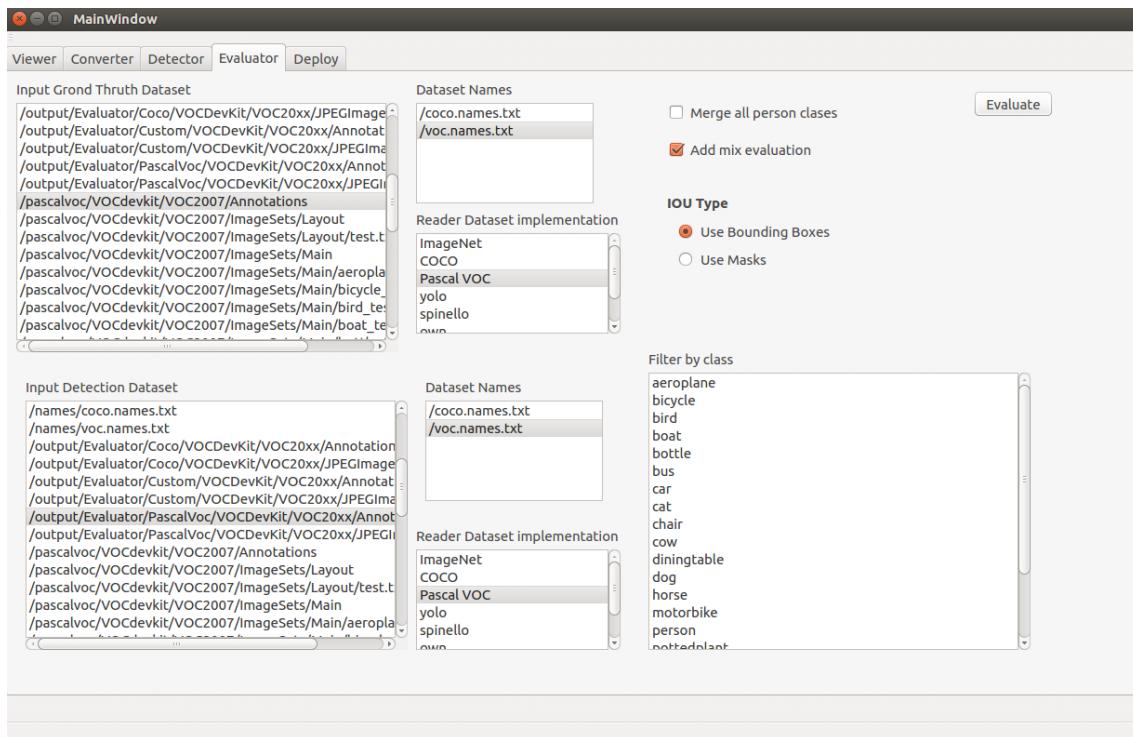


Figura 5.9: Interfaz gráfica de la herramienta *Evaluator*.

Para utilizar esta herramienta es necesario definir varios parámetros como *Input Ground Thruth Dataset* e *Input Detection Dataset*, que especifican el directorio dónde se encuentran las anotaciones reales del conjunto de datos de test y el directorio de las anotaciones generadas tras la detección respectivamente y *Dataset Names* y *Reader Dataset Implementation* que especifican el fichero con los nombre de los objetos existentes y la

⁷<https://github.com/cocodataset/cocoapi>

⁸<http://cocodataset.org/detection-eval>

5.3. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

base de datos utilizada, tanto para las anotaciones reales como para las detectadas.

Tras esto, se puede comenzar con la evaluación de los modelos. Cuando finaliza la ejecución, la primera métrica que podemos comparar es el tiempo que ha tomado *Detection Suite* para evaluar cada una de las redes utilizadas. La tabla 5.1 muestra una comparativa del tiempo que ha requerido la evaluación de cada uno de los modelos.

	VOC	COCO	Custom
Tiempo de Evaluación [s]	4.90815	6.6475	5.28746

Tabla 5.1: Tiempo requerido para la evaluación de cada modelo.

Se puede observar que el modelo preentrenado con PascalVOC ha sido el que más rápido ha realizado la evaluación, siendo el modelo preentrenado con COCO el más lento. El modelo entrenado con la base de datos personalizada se encuentra en medio aunque muy cercano en tiempos al modelo PascalVOC. Teniendo en cuenta las 4952 imágenes del conjunto de test, se puede considerar que la evaluación se ha realizado en tiempos óptimos, sin existir, en este caso, gran diferencia entre los modelos, ya que entre el más lento y el más rápido la diferencia es de 1.7 segundos.

Otras métricas que se muestra por terminal cuando el proceso finaliza son el mAP y mAR conjunto para todas las clases utilizando los 10 umbrales comentados anteriormente del IoU. Además, se muestra el mAP y mAR medio de estos umbrales. La tabla 5.2 muestra el mAP para cada uno de los umbrales mencionados.

Umbrales IoU										
	0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95
VOC	0.488	0.470	0.439	0.396	0.345	0.280	0.197	0.116	0.043	0.003
COCO	0.495	0.476	0.442	0.401	0.349	0.277	0.194	0.112	0.035	0.002
Custom	0.810	0.770	0.693	0.627	0.569	0.547	0.491	0.432	0.387	0.358

Tabla 5.2: mAP conjunto para 10 umbrales de IoU de los 3 modelos evaluados.

Una tendencia común de los 3 modelos es que a media que aumenta el umbral de IoU,

el valor de mAP va disminuyendo. Comparando cada modelo, en primer lugar, se observa unas métricas bastante similares en los 2 modelos preentrenados, teniendo el modelo de COCO mejores valores de mAP en los umbrales de IoU menores a 0.75, siendo superado por el modelo PascalVOC en los umbrales superiores. Para el modelo entrenado se observa gran mejora en los valores de mAP, tanto en los valores bajos del umbral de IoU como en los altos.

Tras el análisis del mAP, se mostrará un comparativa del mAR para cada uno de los 10 umbrales de IoU. La tabla 5.3 muestra estos datos:

	Umbrales IoU									
	0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95
VOC	0.708	0.693	0.669	0.629	0.582	0.516	0.422	0.309	0.171	0.039
COCO	0.736	0.721	0.691	0.653	0.603	0.528	0.432	0.315	0.161	0.030
Custom	0.874	0.857	0.831	0.797	0.748	0.682	0.621	0.565	0.486	0.403

Tabla 5.3: mAR conjunto para 10 umbrales de IoU de los 3 modelos evaluados.

Al igual que con el mAP, se observa una disminución de los valores a medida que aumenta el valor del umbral, aunque los valores para cada uno de ellos son superiores a los calculados del mAP. Comparando los modelos preentrenados, COCO obtiene mejores valores que PascalVOC excepto en los umbrales 0.9 y 0.95. En cuanto al modelo entrenado, al igual que para el mAP, obtiene considerables mejoras respecto a los modelos preentrenados. Finalmente, la tabla 5.4 muestra el valor medio conjunto de los 10 umbrales de IoU, tanto de mAP como de mAR, para cada uno de los modelos evaluados.

IoU=0.5:0.95		
	mAP	mAR
VOC	0.278	0.473
COCO	0.278	0.487
Custom	0.568	0.686

Tabla 5.4: Comparativa conjunta de mAP y mAR de la media de los 10 umbrales de IoU para cada uno de los 3 modelos evaluados.

5.3. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

En las métricas conjuntas para todos los umbrales se observa que para el mAP los modelos preentrenados de PascalVOC y COCO tienen el mismo valor, mientras que el modelo entrenado tiene una mejora del 29 % con respecto a ambos. En cuanto al mAR, el modelo preentrenado de COCO tiene un valor ligeramente superior al de PascalVOC, siendo superado por el modelo entrenado en un 19.9 % y un 21.3 % respectivamente.

En el fichero CSV, comentado anteriormente, que devuelve la herramienta *Evaluator* cuando finaliza su ejecución, se pueden observar los valores de mAP y mAR pero individualmente para cada uno de los 20 objetos sobre los que se ha realizado la detección. Las tablas 5.5 y 5.6 muestran los valores de mAP y mAR respectivamente para cada objeto.

	Objetos																			
	aero	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	person	plant	sheep	sofa	train	tv
VOC	0.32	0.35	0.24	0.15	0.09	0.41	0.32	0.46	0.12	0.27	0.22	0.36	0.40	0.34	0.24	0.10	0.25	0.26	0.41	0.26
COCO	0.33	0.30	0.24	0.15	0.10	0.42	0.32	0.45	0.14	0.33	0.16	0.38	0.39	0.31	0.25	0.10	0.27	0.25	0.39	0.27
Custom	0.62	0.70	0.54	0.42	0.39	0.70	0.61	0.72	0.40	0.61	0.47	0.65	0.66	0.64	0.58	0.38	0.54	0.55	0.69	0.54

Tabla 5.5: mAP medio de los 10 umbrales de IoU para cada uno de los objetos en las 3 redes evaluadas.

	Objetos																			
	aero	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	person	plant	sheep	sofa	train	tv
VOC	0.53	0.50	0.41	0.33	0.20	0.62	0.52	0.65	0.30	0.48	0.45	0.60	0.58	0.50	0.46	0.26	0.43	0.53	0.63	0.52
COCO	0.53	0.48	0.41	0.32	0.25	0.65	0.55	0.65	0.33	0.53	0.48	0.61	0.56	0.51	0.48	0.27	0.46	0.57	0.63	0.48
Custom	0.74	0.69	0.64	0.55	0.46	0.82	0.74	0.84	0.52	0.72	0.67	0.80	0.75	0.71	0.65	0.51	0.65	0.73	0.80	0.73

Tabla 5.6: mAR medio de los 10 umbrales de IoU para cada uno de los objetos en las 3 redes evaluadas.

Tanto los valores de mAP como de mAR de cada objeto se encuentran alrededor de la media general mostrada en la tabla 5.4 sin haber demasiadas diferencias entre los valores más altos y los más bajos. Una característica común, es que, tanto los objetos con valores altos como bajos de mAP y mAR, mantienen su posición en cada uno de los 3 modelos evaluados. Por ejemplo, la detección de gatos representa uno de los valores más altos para cada modelo, mientras que la detección de botellas representa valores bajos en relación con los demás objetos.

Capítulo 6

Conclusiones

En este capítulo se van a exponer las principales conclusiones obtenidas durante el desarrollo de este trabajo. Además, se propondrán varias líneas de trabajo futuras para continuar con la tarea realizada.

6.1. Conclusiones

El desarrollo de este trabajo ha permitido llegar a una serie de conclusiones relacionadas con el aprendizaje profundo utilizando redes neuronales artificiales entrenadas usando la herramienta Caffe. Además, la extracción de varios estadísticos al testear estas redes neuronales, ha permitido medir de manera objetiva las prestaciones de éstas y sacar conclusiones de cómo actúan en la detección de objetos. Las conclusiones serán expuestas siguiendo los *sub-objetivos* expuestos en la sección 2.1.

Estudio de la plataforma Caffe

La plataforma Caffe es una herramienta de gran utilidad para el trabajo con redes neuronales y uno de los frameworks con mayor uso y reconocimiento actualmente. La forma tan sencilla con la que es posible realizar el entrenamiento de un modelo, hace que sea una herramienta muy intuitiva, reforzando su uso frente a otros. Además, aporta varios ejemplos relacionados con la clasificación y detección de objetos, siendo un buen punto de partida para introducirse en el mundo de la inteligencia artificial.

Estudio de varias bases de datos aplicadas a la detección

Con el estudio de PascalVOC y COCO, se ha podido comprobar cuáles son los

6.1. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

puntos que definen y diferencian a las bases de datos creadas para el entrenamiento de redes neuronales dedicadas a la detección. La cantidad de imágenes que las componen y cómo están distribuidos los objetos presentes en ellas, las anotaciones y etiquetado de los objetos de estas imágenes y el formato de éstas y el número de clases de objetos existentes en ellas. Además, con las pruebas realizadas se ha podido comprobar como se comportan redes entrenadas con ellas.

Estudio de la técnica SSD

Se exponen las principales características de esta técnica, tanto en el modelo que se propone como a la hora del entrenamiento. Se explica como la utilización de cajas delimitadoras con múltiples escalas conectadas con mapas de características permite modelar las diferentes formas de los objetos. Además, la utilización de un modelo con varias capas convolucionales permite que la imagen de entrada disminuya de tamaño paulatinamente, obteniendo así características en múltiples escalas y generando salidas más precisas. Los desarrolladores de SSD integran esta técnica con Caffe, exponiendo varios ejemplos y redes preentrenadas, que son un buen punto de partida para conocer los resultados que da un modelo que utiliza esta técnica. Además, la integración con este framework hace posible el entrenamiento de un modelo SSD para su posterior estudio.

Desarrollo de un componente detector

El componente Python desarrollado ofrece buenos resultados con las redes neuronales entrenadas previamente. El desacoplar la tarea en 3 hilos, dedicando uno únicamente al proceso de detección, hace que la aplicación sea más robusta evitando que ésta baje su rendimiento debido a la detección, ya que se realiza a la velocidad que pueda soportar la CPU del equipo utilizado. Se trata del primer contacto a la hora de comparar el rendimiento de las redes utilizadas ya que se puede observar de manera visual la capacidad que tienen los modelos para detectar objetos.

Creación de una base de datos personalizada

Se ha propuesto un *script* escrito en Python para la creación de bases de datos personalizadas a partir de ficheros LMDB. De esta manera, se podrán crear conjuntos de imágenes en función de las necesidades que se requieran y utilizarlas para entrenar, validar o testear una red neuronal.

Entrenamiento de una red neuronal

Se han definido los puntos necesarios para entrenar una red neuronal en Caffe. En primer lugar, la estructura de la red neuronal sobre la que se ha realizado el entrenamiento, dónde se puede comprobar cada una de las capas existentes y el funcionamiento de cada uno de ellas. Posteriormente, se explica el funcionamiento del solucionador del entrenamiento en Caffe, con el que se pueden conocer los parámetros con los que se va a entrenar la red y poder ajustarlos en función de las necesidades. Finalmente, se expone el *script* necesario para ejecutar el entrenamiento, dónde sale a relucir la facilidad con la que es posible entrenar un modelo en Caffe.

Realización de experimentos con la herramienta *Detection Suite* de JdeRobot

Se han explicado las métricas que calcula *Detection Suite* y que se han utilizado para evaluar los modelos. Esta introducción ha hecho que fuera más fácil interpretar posteriormente los resultados obtenidos. Después, con la utilización de *Detection Suite* se han podido comprobar varias de las funcionalidades que tiene esta herramienta. Primero, con el detector, se ha comprobado como de una forma muy sencilla, se puede realizar la detección de objetos sobre un gran conjunto de datos. Después, con el evaluador, se ha verificado que con *Detection Suite* de una forma muy sencilla e intuitiva, se pueden calcular diferentes métricas de varios modelos. Finalmente, evaluando los datos obtenidos, se ha podido verificar que con una rede neuronal entrenado se obtienen mejores valores tanto de mAP como de mAR, mejorando estos valores en alrededor de un 30 % y un 20 % respectivamente.

6.2. Líneas futuras

Tras la exposición de las conclusiones, se enumeran varias líneas de investigación que ayudarían a continuar con lo desarrollado durante este trabajo.

- Una línea de investigación cercana a la abordada sería la de detección de objetos utilizando algunos de los softwares adicionales para redes neuronales explicados en el apartado 1.2.3. De esta manera es posible contrastar como funciona cada framework y elegir cualquiera de ellos en función de lo requerido.

6.2. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

- Dado que *Detection Suite* soporta otras bases de datos, como YOLO[34] o ImageNet[35], sería interesante utilizarlas para entrenar una red y así ver los resultados que genera la utilización de cada una de ellas.
- Otro punto interesante a abordar, sería el etiquetado de imágenes para crear una propia base de datos y entrenar, validar y testear una red con ella.
- En relación a las pruebas realizadas, se podría usar la interfaz gráfica de *Detection Suite* utilizando vídeos grabados previamente o las imágenes captadas por una cámara en tiempo real. Así, se puede comprobar como actúan las redes que se evalúen detectando objetos en movimiento.
- En relación con los datos obtenidos tras la evaluación de los modelos, se podría realizar un análisis de todo lo necesario para entrenar una red, es decir, base de datos para el entrenamiento y validación, estructura de la red neuronal y parámetros del solucionador, para intentar crear un modelo más robusto y mejorar las métricas obtenidas en el apartado 5.3.

Bibliografía

- [1] Wei Di, Anurag Bhardwaj, Jianing Wei Deep Learning Essentials. Your hands-on guide to the fundamentals of deep learning and neural network modeling
- [2] John Wiley & Sons. Encyclopedia of Artificial Intelligence
- [3] Raúl Pino Diez, Alberto Gómez Gómez, Nicolás de Abajo Martínez. Introducción a la Inteligencia Artificial. Sistemas expertos, redes neuronales artificiales y computación evolutiva Noviembre 2003
- [4] Francisco Escolano, Miguel Ángel Cazorla, María Isabel Alfonso, Otto Colomina y Miguel Ángel Lozano. Inteligencia Artificial: Modelos, Técnicas y Áreas de Aplicación Noviembre 2003
- [5] Aprendizaje automático. https://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico
- [6] D. Wang, A. Khosla, R. Gargaya, H. Irshad, and A. H. Beck. Deep Learning for Identifying Metastatic Breast Cancer. *ArXiv e-prints.*, June 2016.
- [7] Edgar Nelson Sánchez Camperos y Alma Yolanda Alanís García. Redes Neuronales: Conceptos fundamentales y aplicaciones a control automático
- [8] La predicción del dato: Redes Neuronales Artificiales. <https://www.analiticaweb.es/la-prediccion-del-dato-redes-neuronales-artificiales/>
- [9] Pedro Larrañaga, Iñaki Inza, Abdelmalik Moujahid. Departamento de Ciencias de la Computación e Inteligencia Artificial. Universidad del País Vasco?Euskal Herriko Unibertsitatea. Tema 8. Redes Neuronales <http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/t8neuronales.pdf>
- [10] Jordi Torres. Deep Learning - Introducción práctica con Keras <https://jorditorres.org/redes-neuronales-densamente-conectadas/>

6.2. APLICACIÓN PARA DETECCIÓN DE OBJETOS CON SSD

- [11] DEWI SURYANI, S.KOM., M.ENG. CONVOLUTIONAL NEURAL NETWORK, *Binus University - School of Computer Science.* <http://soc.s.binus.ac.id/2017/02/27/convolutional-neural-network/>
- [12] Caffe, Deep Learning Framework. <https://caffe.berkeleyvision.org/>
- [13] TensorFlow, An end-to-end open source machine learning platform. <https://www.tensorflow.org/>
- [14] Keras: The Python Deep Learning library. <https://keras.io/>
- [15] Darknet: Open Source Neural Networks in C <https://pjreddie.com/darknet/>
- [16] Machine Learning and Deep Learning All the solutions you need to engage your customers wherever they are. <https://www.argility.com/argility-ecosystem-solutions/iot/machine-learning-deep-learning/>
- [17] Deep Learning. Deep Learning (Aprendizaje profundo). Tres cosas que es necesario saber. <https://es.mathworks.com/discovery/deep-learning.html>
- [18] Nuria Oyaga de Frutos. Análisis de Aprendizaje Profundo con la plataforma Caffe. Trabajo Fin de Grado. Escuela Técnica Superior de Ingeniería de Telecomunicación. Universidad Rey Juan Carlos. Curso Académico 2016/2017.
- [19] David Pascual. Deep Learning on RGBD sensors. Trabajo Fin de Grado. Escuela Técnica Superior de Ingeniería de Telecomunicación. Universidad Rey Juan Carlos. Curso Académico 2016/2017.
- [20] Nacho Condés. Deep Learning Applications for Robotics using TensorFlow and JdeRobot. Trabajo Fin de Grado. Escuela Técnica Superior de Ingeniería de Telecomunicación. Universidad Rey Juan Carlos. Curso Académico 2016/2017.
- [21] Wei Liu ,Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed , Cheng-Yang Fu, Alexander C. Berg. Single Shot MultiBox Detector
- [22] Caffe, Deep Learning Framework. Interfaces. <http://caffe.berkeleyvision.org/tutorial/interfaces.html>
- [23] Caffe, Deep Learning Framework. Layer Catalogue. <https://caffe.berkeleyvision.org/tutorial/layers.html>

- [24] JdeRobot. an open source toolkit for Robotics and Computer Vision!. https://jderobot.org/Main_Page
- [25] The PASCAL Visual Object Classes Homepage. <http://host.robots.ox.ac.uk/pascal/VOC/>
- [26] PASCAL VOC2012 Database Statistics <http://host.robots.ox.ac.uk/pascal/VOC/voc2012/dbstats.html>
- [27] COCO, Common Objects in Context. <http://cocodataset.org/#home>
- [28] COCO, Common Objects in Context. Data format. <http://cocodataset.org/#format-data>
- [29] Protocol Buffers. Developer Guide. <https://developers.google.com/protocol-buffers/docs/overview>
- [30] Single Shot MultiBox Detection. Understanding SSD MultiBox - RealTime Object Detection In Deep Learning. <https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fa>
- [31] Simonyan,K.,Zisserman,A.: Simonyan,K.,Zisserman,A.:Very deep convolutional networks for large-scale image recognition. In: NIPS. (2015)
- [32] Erhan, D., Szegedy, C., Toshev, A., Anguelov, D.: Scalable object detection using deep neural networks. In: CVPR. (2014)
- [33] Índice Jaccard. https://en.wikipedia.org/wiki/Jaccard_index
- [34] YOLO: Real-Time Object Detection. <https://pjreddie.com/darknet/yolo/>
- [35] ImageNet. <http://www.image-net.org/>