CURSO 2017/2018

"DETECCIÓN DE OBJETOS A TIEMPO REAL CON UAVs MEDIANTE APRENDIZAJE PROFUNDO: Aplicación al reconocimiento de señales de tráfico"

DAVID SÁNCHEZ FRAILE

MÁSTER EN INGENIERÍA DE SISTEMAS Y CONTROL

TRABAJO FIN DE MÁSTER

4 DE SEPTIEMBRE DE 2018

DIRECTOR: GONZALO PAJARES MARTINSANZ

MÁSTER EN INGENIERÍA DE SISTEMAS Y CONTROL

PROYECTO ESPECÍFICO PROPUESTO POR EL ALUMNO

"Detección de objetos a tiempo real con UAVs mediante aprendizaje profundo: Aplicación al reconocimiento de señales de tráfico"

Autor: David Sánchez Fraile

Director: Gonzalo Pajares Martinsanz

Autorización de difusión

Autorizamos a la Universidad Complutense y a la Universidad Nacional de Educación a Distancia a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado el autor:

David Sánchez Fraile

4 de Septiembre 2018

Abstract

This Master Thesis' work describes the steps followed to develop a ROS (Robotic Operating System) based system which provides us with a remote control to pilot ARDrone and at the same time performs real-time object detection with Deep Learning techniques, subscribing to the images coming from the drone with a Convolutional Neural Network processing node. Two models are studied: SSD Mobilenet and YOLO. The thesis describes the system integration steps as well as a detailed study of the results obtained in different tests performed that begins with a simple Virtual Machine system and progresses to a system that uses a GPU Jetson TX2 embedded computer and ROS image compression. The work also approaches the fine-tuning problem in Convolutional Neural Networks. We re-train both proposed models with traffic signal images and finally compare the studied models in speed, detection confidence and hardware performance terms.

Keywords

UAV, ROS, Deep Learning, Convolutional Neural Network, SSD, YOLO, OpenCV, TensorFlow, GPU, Real-Time detection.

Resumen

Este Trabajo Fin de Máster describe el proceso llevado a cabo para la creación de un sistema basado en ROS (Robotic Operating System) que permite, por un lado el pilotaje remoto del dron ARDrone 2.0 y por otro lado la detección de objetos a tiempo real mediante técnicas de aprendizaje profundo gracias a la subscripción a las imágenes del dron con un nodo que las procesa haciendo uso de redes neuronales convolucionales, concretamente se implementan dos modelos distintos, SSD Mobilenet y YOLO. El trabajo recoge tanto los pasos seguidos para integrar el sistema, como un detallado estudio de los resultados obtenidos en diversas pruebas, que comienzan con un modelo basado en máquinas virtuales hasta llegar a la integración con un sistema embebido como la GPU Jetson TX2 y comprimiendo las imágenes en ROS para optimizar el procesado de imágenes para la detección de objetos. Asimismo el trabajo aborda el problema del reentrenamiento de redes neuronales para la aplicación de los modelos originales a la detección de señales de tráfico y finalmente compara los modelos estudiados en términos de velocidad, porcentaje de confianza en la detección y rendimiento del hardware utilizado.

Palabras clave

UAV, ROS, Aprendizaje profundo, Red Neuronal Convolucional, SSD, YOLO, OpenCV, TensorFlow, GPU, Detección a Tiempo Real.

Índice general

Lista de Figuras

Lista de Tablas

Agradecimientos

Me gustaría agradecer profundamente y en primer lugar a mi tutor, Gonzalo Pajares, su confianza desde los primeros pasos que di en el área de procesado de imágenes y visión por computador. Su apoyo, entusiasmo y experiencia me han motivado enormemente durante el estudio de este Máster y me han guiado para el desarrollo del presente Trabajo Final, que lo culmina.

Mi más sincero agradecimiento a Eva Besada, por su confianza inicial, así como a todos los profesores que han compartido su conocimiento conmigo durante estos años de estudio de postgrado. Hoy, puedo decir que sé un poquito más que ayer.

A mi familia por estar ahí, como siempre, con su apoyo incondicional.

A Patricia, por su paciencia, por estar ahí, porque sin ella, esto no habría sido posible.

Abreviaturas

TFM	Trabajo Fin de Máster
VANT	Vehículo aéreo no tripulado
UAV	Unmanned aerial vehicle
GPS	Global Positioning System
DL	Deep Learning
GPU	Graphics Processing Unit
RNA	Red Neuronal Artificial
FC	Capa totalmente conectada *(Full Connected)*
CNN	Red Neuronal Convolucional *(Convolutional Neural Network)*
SSD	Single Shot Multibox Detector
R-CNN	Redes neuronales convolucionales basadas en region *(Region-based convolutional neural networks)*
R-FCN	Redes plenamente convolucionales basadas en región *(Region-based Fully Convolutional Networks)*
SVM	Support Vector Machine
ROI	Región de interés (Region of Interest)
RPN	Red de propuesta de regiones. *(Region Proposal Network)*
IOU	Ratio de Unión sobre Intersección *(Intersection over Union)*
NMS	Supresión no máxima (*Non-Maximum Suppression)*
YOLO	You Only Look Once
GT	Ground Truth, Gt (Imagen de referencia)
CUDA	Computer Unified Device Architecture
FPS	Frames por Segundo
API	Interfaz de Programación de Aplicaciones (Application Programming Interface)

1. Capítulo 1. Introducción

Un vehículo aéreo no tripulado (VANT) o UAV (del inglés *Unmanned Aerial Vehicle*), llamado de forma común dron, es una aeronave que no lleva piloto a bordo. Los UAVs pueden ser pilotados de forma remota o bien pueden volar de forma autónoma, gracias a planes de vuelo pre-programados o gracias a un sistema más complejo de automatización dinámica (THE UAV (2018)).

Para poder controlar los UAVs con cualquier método referenciado en el párrafo anterior, esto es, pilotaje remoto o bien vuelo autónomo, los sensores instalados a bordo se hacen fundamentales. Algunos de los sensores que pueden instalarse a bordo son sensores de ultrasonidos, láser, cámaras, GPS, entre otros. Evidentemente los sistemas de control de esos sensores y de los motores que permiten la propulsión de estos dispositivos son clave tanto para el pilotaje de las aeronaves como para la obtención de información y datos que pueden permitir diversas aplicaciones para los que pueden destinarse los drones: vigilancia y control, rescate de personas, envío de paquetería, geolocalización, aplicaciones agrícolas, repoblación forestal entre otras muchas (Pajares, 2015).

Este Trabajo Fin de Máster (TFM) se centra en la consecución de pilotaje remoto y en la obtención de imágenes de la cámara frontal de un dron, mediante la integración de diversas tecnologías y desarrollos hardware y software, constituyendo uno de los pilares del trabajo. Por otra parte, a partir de las imágenes obtenidas se pretende llevar a cabo un procesado a partir de técnicas actuales de detección de objetos tratando de llegar a conseguir procesado a tiempo real. El área de aplicación fundamental es, por tanto la visión por computador y dentro de este área nos centramos en Deep Learning (DL), encuadrada dentro del más amplio campo del Machine Learning, aunque también se trabajan otros campos como la transmisión de datos, redes, etc.

El proyecto se plantea desde un punto de vista eminentemente práctico, utilizando hardware real y tratando de utilizar software lo más actual posible, como se verá a lo largo del TFM.

1.1. Objetivos

Este TFM desarrolla un sistema que permite pilotar un UAV de forma remota mientras se obtienen y procesan las imágenes de una de sus cámaras para llevar a cabo detección de objetos en un ordenador remoto. Los objetivos que se plantean en el proyecto son los siguientes:

I. Conseguir un sistema de pilotaje remoto del dron mediante la integración de diversas tecnologías hardware-software.

II. Estudiar dos alternativas a la detección de objetos a tiempo real con las imágenes obtenidas del dron.

III. Comparar las dos alternativas a la detección de imágenes para una serie de clases en distintas condiciones:

- Con compresión de imagen y sin ella.
- Llevando a cabo el procesado de imágenes en un sistema embebido con GPU (Graphics Processing Unit).

IV. Reentrenar ambos modelos con imágenes de señales de stop y ceda el paso, aplicándolos a la detección de sobre las imágenes del dron.

1.2. Descripción del trabajo desarrollado

Este proyecto integra un sistema basado en ROS (Robotic Operating System) que permite controlar un dron de forma remota mientras se reciben imágenes del mismo a partir de las cuales se lleva a cabo detección de objetos mediante técnicas de aprendizaje profundo.

Se parte de un sistema para controlar el dron que ya existía, pero no estaba integrado en ROS, así pues, el primer paso para el desarrollo del proyecto fue ese,

integrar el driver de control de navegación remota del dron en ROS para la obtención de imágenes del mismo, mediante trabajo personal y colaborativo.

El siguiente paso consiste en lograr que el dron detecte objetos haciendo uso de redes neuronales convolucionales, concretamente se trabaja con el modelo SSD Mobilenet. La detección se lleva a cabo en un nodo ROS que suscribe al tópico que toma las imágenes del dron, obteniendo un nuevo tópico con la salida de la red neuronal. Dado que las primeras pruebas dan como resultado una baja velocidad de proceso, al tener claras limitaciones de hardware por trabajar con máquinas virtuales en un mismo ordenador portátil, se decide dar un salto de calidad en lo que respecta a hardware e integrar el sistema embebido *Jetson TX2 de Nvidia*.

Tras lograr integrar la Jetson TX2 en el sistema y hacer una serie de pruebas se observa que la velocidad de proceso aún no es la esperada, descubriendo un "cuello de botella" en el envío de imágenes de la máquina virtual en la que se ejecuta el driver de control del dron hasta la GPU. Se resuelve el problema comprimiendo las imágenes mediante la creación de dos tópicos intermedios, uno que comprime las imágenes originales procedentes del dron y otro que las descomprime, ya en la Jetson TX2.

Finalmente, una vez integrado el sistema completo con la GPU y las imágenes comprimidas, se realizan distintas pruebas con SSD Mobilenet y con otro modelo de red neuronal convolucional de similares características y directa competencia, YOLO. Las pruebas se realizan con las clases de los modelos originales y también con modelos reentrenados para la detección de señales de tráfico, concretamente para las clases señal de stop y señal de ceda el paso. Se analizan y comparan los resultados obtenidos en lo que respecta a velocidad de proceso, nivel de confianza en la detección y rendimiento del hardware utilizado.

Los resultados obtenidos son realmente positivos, dado que se logra aumentar la velocidad de proceso de poco más de dos imágenes por segundo (frames por segundo, fps) en la primera prueba a más de veinte fps en la última. Así, el uso del sistema embebido Jetson TX2 y de la compresión de imágenes para su transporte, se convierten en la clave del éxito de los resultados del proyecto planteado.

1.3. Descripción de los capítulos

Este proyecto se divide en cinco capítulos:

1. **Capítulo 1. Introducción.** En este capítulo introductorio se plantea el problema, los objetivos a alcanzar en la resolución del mismo, así como la organización del TFM.

2. **Capítulo 2. Redes neuronales artificiales.** En este capítulo se estudian de forma breve los fundamentos teóricos básicos de las redes neuronales, se estudia el estado del arte en la detección de objetos y se describe la arquitectura de las redes neuronales utilizadas.

3. **Capítulo 3. Integración del sistema.** En este apartado se describe de forma amplia los pasos seguidos para el desarrollo del proyecto, detallando métodos, hardware, software y scripts utilizados.

4. **Capítulo 4. Resultados obtenidos.** Este capítulo recoge los principales resultados obtenidos de los experimentos realizados.

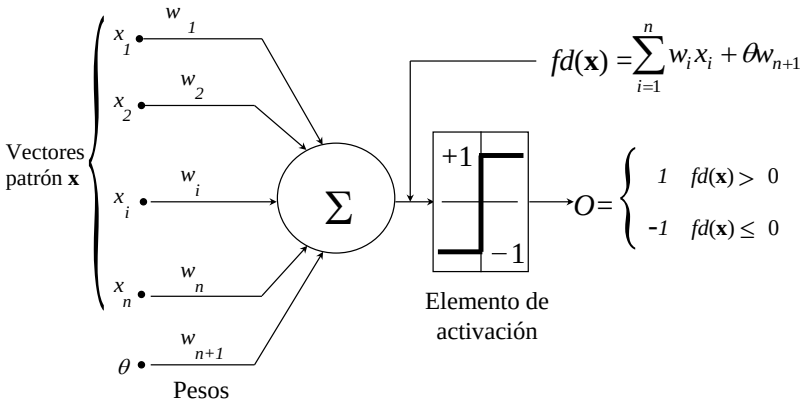
5. **Capítulo 5. Conclusiones y trabajo futuro.** Este apartado recoge las conclusiones extraídas de las pruebas realizadas. Asimismo se hace un planteamiento para la ampliación del proyecto en el futuro.

2. Capítulo 2. Redes Neuronales Artificiales.

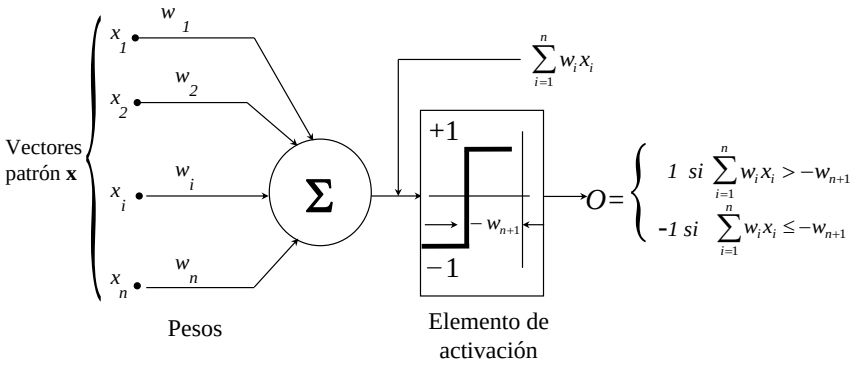
Una red neuronal artificial (RNA) es un modelo matemático que se inspira en el comportamiento biológico que presentan las neuronas en un cerebro. De entre las muchas definiciones que se pueden encontrar en la bibliografía, podemos destacar la de *Haykin (1999)*, que define red neuronal como un sistema de procesado de información distribuido masivamente y que tiene ciertas características semejantes a las redes neuronales biológicas del cerebro humano.

Los primeros avances datan de los años cincuenta y sesenta, cuando el científico Frank Rosenblatt, inspirado por los trabajos previos de Warren MCCulloch y Walter Pitts, desarrolló los perceptrones (Pajares, de la Cruz, 2007).

Actualmente no es común usar redes neuronales basadas en perceptrones, el principal modelo usado que se usa es la neurona sigmoidea (*Nielsen, 2017)*. Como cita *Nielsen (2017)* la neurona sigmoidea puede tomar valores de entrada entre 0 y 1, a diferencia del perceptrón que sólo puede tomar valores de entrada binarios, . La neurona tiene pesos para cada entrada (w1, w2…w_n) y un *bias* general, además lleva asociada una función de activación, que limita la amplitud de salida de la misma (típicamente en el intervalo [0-1]). La figura 1 muestra el elemento de activación de una neurona, en este caso, un perceptrón. (Pajares y de la Cruz, 2007).



(a)



(b)

Figura 1. Representaciones equivalentes del modelo del perceptrón para dos clases. (Pajares y de la Cruz, 2007).

La arquitectura de una red neuronal suele constar de una capa de entrada, donde se ubican las neuronas de entrada, una zona intermedia en la que se ubican las capas ocultas y una capa de salida. (*Nielsen, 2017).*

Las redes neuronales en las que la salida de una capa se usa como la entrada de la siguiente se denominan *redes de propagación hacia delante*. Las redes en que las neuronas pueden estar conectadas indistintamente con neuronas de niveles previos o posteriores de su mismo nivel se denominan *redes de propagación hacia atrás o recurrentes* (*Nielsen, 2017)*

La figura 2 muestra la arquitectura básica de una red neuronal de retropropagación.

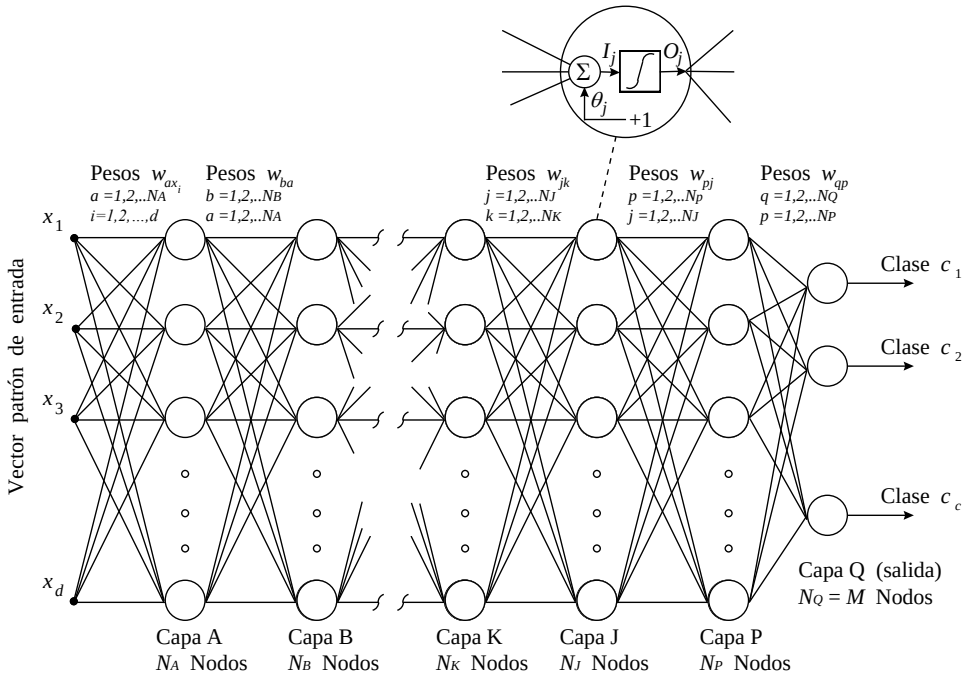


Figura 2. Modelo de red neuronal multicapa retropropagación (Pajares y de la Cruz, 2007).

La red de la figura 2 consta de capas de neuronas de cómputo estructuradas de forma idéntica y colocadas de manera que la salida de cada neurona en una capa proporciona la entrada de cada neurona en la siguiente capa (Pajares y de la Cruz, 2007)

Uno de los tipos de redes con capacidad para realizar lo que se conoce como aprendizaje profundo se denomina red neuronal convolucional (CNN, Convolutional Neural Network). Las redes neuronales convolucionales son redes multicapa que se componen de capas convolucionales y de sub-muestreo alternadas, presentando al final capas totalmente conectadas (Dertat, 2017). La figura 3 muestra un esquema general y habitual de una CNN, cuyos componentes se explican a continuación

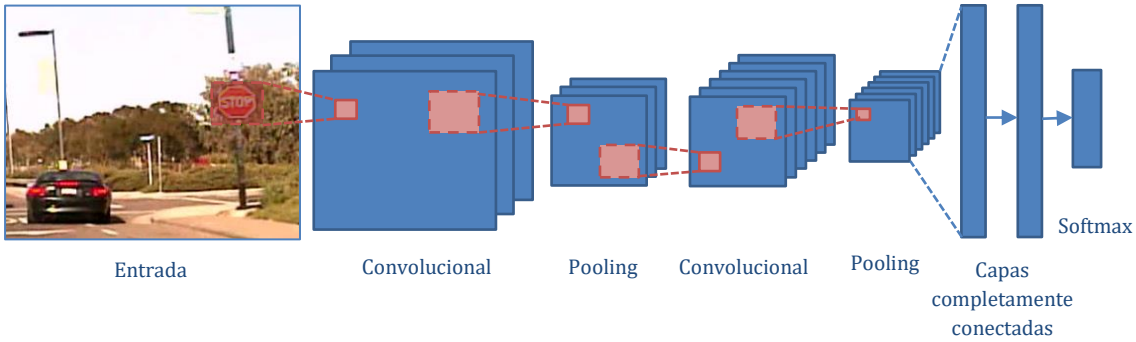


Figura 3. Arquitectura habitual de una red CNN.

CAPAS CONVOLUCIONALES

La convolución es una operación matemática que permite la unión de dos conjuntos de información. En el ejemplo mostrado en la figura 4 la convolución se aplica a los datos de entrada usando un filtro que realiza lo que se conoce como operación de convolución para producir un mapa de características. (Pajares y de la Cruz, 2007).

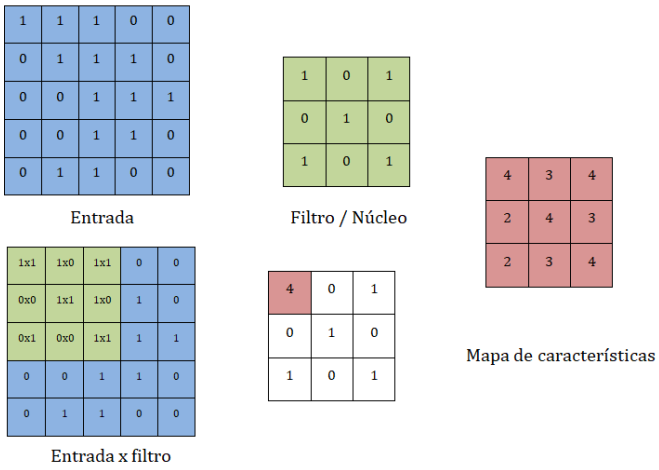


Figura 4. Operación de convolución.

Se lleva a cabo la operación de convolución desplazando el filtro a lo largo de la imagen de entrada, por ejemplo de izquierda a derecha y de arriba hacia abajo. En cada localización se realiza el producto de la matriz para finalmente sumar el resultado. Esta suma pasa a formar parte del mapa de características. La zona verde donde se lleva a cabo la convolución se denomina *campo de recepción.* Dado el tamaño del filtro de la figura 4, el *campo de recepción* es de 3x3. Esta operación se completa deslizando el filtro por toda la matriz de entrada hasta al final obtener el mapa de características que se muestra en la parte de la derecha de la imagen de la figura 4.

CAPAS DE POOLING

La capa de Pooling, o reducción, disminuye la cantidad de parámetros, al quedarse con las características más comunes. Esto se lleva a cabo extrayendo características esenciales, algunas de las cuales son el promedio o el máximo de una región. Por ejemplo en la figura 5 se realiza una operación de máximo para obtener el resultado mostrado en la parte derecha (Dertat, 2017)

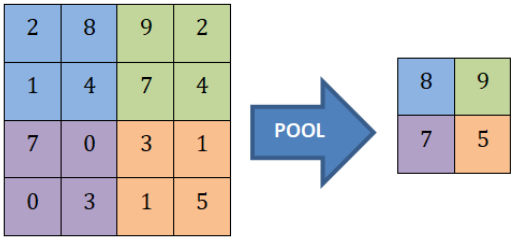


Figura 5. Operación de la capa POOLING

Como se puede comprobar en la figura 6, el mapa de características del que partíamos, con dimensiones 32x32x10 acaba con dimensiones 16x16x10. Tras haber aplicado la operación de pooling, la profundidad del mapa de características no se ve afectada.

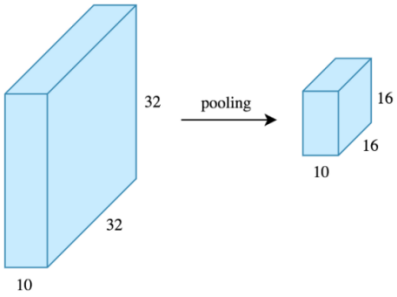


Figura 6. Operación de la capa Pooling. (Dertat, 2017)

CAPAS TOTALMENTE CONECTADAS (FC)

Tras las capas de convolución y de pooling, se añaden dos capas totalmente conectadas para completar la arquitectura de la CNN.

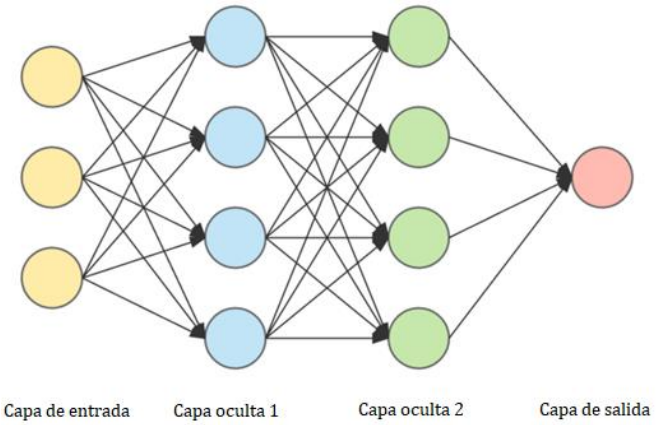


Figura 7. Capas totalmente conectadas

Como comenta Dertat, (2017), los problemas más habituales en el uso de redes neuronales aparecen en la fase de entrenamiento, que constituye una etapa clave junto con la de decisión, ambas presentes en todo proceso de aprendizaje, incluyendo el aprendizaje profundo. Detecciones pobres o con poca precisión suelen tener su procedencia en una mala ejecución de la fase de entrenamiento. Los principales motivos por los que un determinado proceso de entrenamiento no converge habitualmente son:

- Insuficiente número de muestras.
- Muestras no representativas. No suficiente heterogeneidad en las muestras.

A continuación estudiamos el estado del arte de las CNN en el ámbito de la detección.

2.1. Estado del arte en la detección.

Las nuevas tecnologías tales como vehículos autónomos, video vigilancia inteligente, detección facial y otras aplicaciones similares son cada vez más demandadas y requieren de una mayor precisión y velocidad de proceso. Estos sistemas no sólo requieren de reconocimiento y clasificación de objetos presentes

en la imagen, sino que además se hace necesaria la localización del objeto. Esto hace que la detección de objetos sea una tarea aún más compleja que su predecesora en el ámbito de la visión por computador, la clasificación de imágenes (*Joyce, 2017)*.

Afortunadamente, las mejores aproximaciones a la detección de objetos son, de hecho, extensiones de modelos de clasificación de imágenes. Hace ya más de un año, Google lanzó una nueva API de detección de objetos para TensorFlow *(TENSORFLOW, 2018)*. Con este lanzamiento vino el lanzamiento de nuevas arquitecturas con sus correspondientes pesos *(Joyce, 2017)*.

- Single Shot Multibox Detector (SSD) con MobileNets.
- SSD con Inception V2.
- Region-Based Fully Convolutional Netwoks (R-FCN) con Resnet 101.
- Faster R-CNN con Resnet 101.
- Faster R-CNN con Inception Resnet v2.

A continuación se estudia brevemente cada modelo y en orden cronológico según su aparición.

2.1.1. R-CNN

Como comenta *Joyce (2017)*, las redes neuronales convolucionales basadas en región o R-CNN fueron las que revolucionaron el desarrollo actual. Como se cita en *Girshick y col., 2014),* consisten en tres fases:

1. Captura de la imagen de entrada en búsqueda de posibles objetos usando un algoritmo llamado *Selective Search*, que extrae aproximadamente 2000 regiones de la imagen (*regiones propuestas*).
2. Ejecución de la CNN sobre las regiones propuestas.
3. Se toma la salida de cada una de las CNN y se realimentan a:
    a. Una SVM (*Support Vector Machine*) para clasificar la región (Vega y Dormido-Canto, 2010).
    b. Un regresor lineal para encajar el objeto lo más posible con la caja delimitadora, si es que el objeto existe.

2.1.2. Fast R-CNN.

El siguiente modelo que sigue a R-CNN cronológicamente es Fast R-CNN. Como cita *Girshick (2015)*, esta versión se asemeja a su predecesor en muchos aspectos, si bien mejora la velocidad de detección fundamentalmente en dos aspectos:

1. La extracción de características sobre la imagen se lleva a cabo antes de proponer las regiones de interés, por lo tanto usando una sola CNN sobre el total de la imagen en lugar de las 2000 CNNs en las 2000 regiones sobrepuestas.
2. Se reemplaza el SVM por una capa *softmax*, de esta forma se extiende la red neuronal para predicciones en lugar de crear un nuevo modelo.

Una imagen de entrada y múltiples regiones de interés (RoIs) son las entradas que aportamos a una red totalmente convolucional. Cada región de interés (RoI) se asocia con un mapa de características de tamaño fijo que se proyecta sobre un vector de características por capas totalmente conectadas (FCs). La red tiene dos vectores de salida por región de interés: confianza *softmax* y compensación a la regresión de la caja delimitadora por clase (*bbox regressor*) *(Joyce,2017).*

El modelo Fast R-CNN funciona mucho mejor en términos de velocidad. Sólo quedaba un "cuello de botella" pendiente de resolver: el algoritmo selectivo de búsqueda para generar las regiones propuestas *(Joyce, 2017)).*

2.1.3. Faster R-CNN.

El objetivo principal de Faster R-CNN era reemplazar el lento algoritmo de selección de regiones propuestas con una red neuronal rápida. Para ello se introdujo la red de propuesta de regiones *(Region Proposal Network, RPN) (Ren y col., 2016).*

Como se menciona en *Ren y col. (2016)*, la red RPN realiza el siguiente proceso:

1. En la última capa de una CNN inicial, una ventana deslizante de 3x3 se mueve a lo largo del mapa de características y re-mapea en dimensiones inferiores.
2. Para cada localización de la ventana deslizante, se generan múltiples posibles regiones basadas en un ratio fijo de fijación de cajas (cajas delimitadoras predeterminadas)
3. Cada propuesta de región consiste en:

    a. Una puntuación de "objetividad" para esa región.
    b. Cuatro coordenadas que representan la caja delimitadora de la región.

Las puntuaciones 2k (*2k scores*) representan la confianza *softmax* para cada una de las *k* cajas delimitadoras que están sobre un "objeto". Cabe señalar que, aunque la RPN procesa las coordenadas de los cuadros delimitadores, no clasifica los objetos potenciales, su único propósito es proponer regiones en los que puede encontrarse el objeto. Si la "objetividad" que presenta una determinada caja supera un determinado valor ("puntuación"), las coordenadas de esa caja pasan como una propuesta de región *(Ren y col. 2016)*.

Una vez obtenidas las propuestas de regiones, las alimentamos a través de lo que es esencialmente una *Fast R-CNN*. Se añade una capa *pooling,* algunas capas conectadas y finalmente una capa de clasificación *softmax* y un regresor de cajas delimitadoras. En resumen, podemos considerar **Faster R-CNN** como una suma de **RPN** y **Fast R-CNN** *(Joyce, 2017)*.

Con esta configuración la red Faster R-CNN presenta una velocidad y precisión realmente novedosas, con un rendimiento verdaderamente significativo. De hecho, este es el punto de partida para muchos modelos de detección, incluyendo el que se describe a continuación *(Joyce, 2017).*

2.1.4. R-FCN.

R-FCN es un modelo de detección de objetos mediante redes plenamente convolucionales basadas en región. La motivación inicial para el desarrollo de este modelo, era, al igual que en Faster R-CNN, incrementar la velocidad maximizando la computación compartida *(Joyce, 2017).*

Las R-FCN comparten al 100 % la computación a través de cada salida individual. El hecho de que sea completamente convolucional se plantea como un problema único en el diseño del modelo (*Dai y col. 2016)*.

Por otro lado, como se comenta en *Dai y col. (2016),* cuando se lleva a cabo la clasificación de un objeto, resulta de interés que el modelo sea invariante en la localización sin importar su localización espacial en la imagen. Cuando se lleva a cabo la detección del objeto el modelo debe aprender las variaciones, esto es, se desea delimitar la caja de detección donde se encuentra el objeto. Dado que estamos tratando de compartir computaciones convolucionales, es fundamental el equilibrio entre la invariancia de la clasificación y la variación de la localización que presenta la detección. Para ello, este modelo introduce el concepto de mapas de puntuación de sensibilidad de posición *(position-sensitive score maps).*

Cada mapa de puntuación representa una posición relativa de una clase. Así, cada mapa de puntuación debe activarse al detectar una parte de un objeto. Esencialmente, estos mapas de puntuación no son más que mapas de características que han sido entrenados para reconocer ciertas partes de un objeto.

Una R-FCN funciona de la siguiente forma (*Dai y col. ,2016)*:

1. Se ejecuta una RPN sobre la imagen de entrada.
2. Se agrega una capa completamente convolucional, que incluye todos los mapas generados en el paso anterior para obtener un conjunto de puntuaciones de las imágenes generales (“banco de puntuaciones”). Deberíamos obtener $k^2(C+1)$ mapas de puntuación, siendo k^2 el número que representa las posiciones relativas para dividir un objeto (por ejemplo,

3^2 para una cuadrícula de 3x3) y $(C+1)$ representa el número de clases más el fondo.

3. Se ejecuta una red completamente convolucional basada en la RPN para generar regiones de interés (RoI).
4. Cada región de interés se divide en el mismo número de k^2sub-regiones que mapas de puntuación se tienen.
5. Para cada subregión se comprueba el banco de puntuaciones para comprobar si esa subregión se corresponde con alguna posición de algún objeto. Este proceso se repite para cada clase.
6. Una vez obtenido un valor de correspondencia con un objeto para cada subregión k^2 de cada clase, se obtienen una puntuación individual promedio por clase.
7. Clasificación de los RoI con una función *softmax* en los restantes C+1 vectores dimensionales.

Con esta configuración, R-FCN es capaz de mantener equilibrio entre la variación que presenta la clasificación proponiendo diferentes regiones de objetos y la invariancia en la localización a través de la referenciación de cada región al mismo banco o los mismos mapas de puntuación (*Dai y col. 2016)*.

La mejor parte de esta red es que es plenamente convolucional, esto es, que toda la computación se comparte a lo largo de toda la red. Como resultado, R-FCN es varias veces más rápida que Faster R-CNN y consigue una precisión comparable.

2.1.5. SSD.

Este es el último modelo estudiado relativo a la parte desarrollada sobre TensorFlow, SSD o Single-Shot-Detector. Al igual que sucedía con R-FCN, proporciona una gran velocidad de proceso en comparación con Faster R-CNN, consiguiéndolo de una forma marcadamente diferente *(Joyce, 2017).*

Los primeros modelos estudiados en este apartado llevan a cabo las propuestas de regiones y la clasificación de regiones en dos pasos separados. Primero se usa una

red de propuesta de regiones de interés, después se clasifican estas regiones bien a través de sus capas totalmente conectadas o bien a través de las capas convolucionales de sensibilidad de posición. SSD lleva a cabo estos dos pasos en uno solo ("single shot"), prediciendo simultáneamente la caja de detección y la clase al procesar la imagen.

Concretamente, dada una imagen de entrada y un conjunto de etiquetas para el conjunto de datos, SSD funciona de la siguiente manera *(Liu y col. 2016):*

1. La imagen pasa a través de una serie de capas convolucionales, produciendo mapas de características a diferentes escalas (por ejemplo 10x10, luego 6x6, nuevamente 3x3).
2. Para cada posición en cada uno de esos mapas de características, se usa un filtro convolucional de 3x3 para evaluar un pequeño conjunto de cajas de detección. Estas cajas de detección predeterminadas son esencialmente equivalentes a las cajas de fijación (*anchor boxes*) estudiadas en el modelo Faster R-CNN.
3. Por cada caja, simultáneamente se predice:
    a. La compensación para la caja de detección.
    b. La confianza de detección de clases.
4. Durante el proceso de entrenamiento, se hacen coincidir las cajas de predicción en base al índice de jaccard [Intersection over Union; IoU]. La caja con mejor predicción será etiquetada como positiva, junto con el resto de cajas en las que se obtiene un IoU con valor superior a 0,5.

Como cita Joyce (2017), el planteamiento de SSD parece sencillo, sin embargo, el proceso de entrenamiento de la red constituye un reto único. En los modelos anteriores, la red de propuesta de regiones aseguraba que cada objeto que intentábamos clasificar tenía un mínimo de probabilidad de pertenecer a una clase. Con SSD, sin embargo, obviamos ese paso de filtrado. Se clasifica y dibujan cajas de detección desde cada posición de la imagen, usando diferentes formas múltiples. Como resultado, se genera un número mucho mayor de cajas de detección que en otros modelos, con un alto número de detecciones negativas.

Para evitar este desajuste, SSD aplica dos procesos fundamentalmente (*Liu y col. 2016)*:

1. Primero utiliza un método llamado supresión no máxima [*non-maximum suppression (NMS)]* cuyo objetivo es agrupar varias cajas superpuestas en un único cuadro. Por ejemplo, si tenemos cuatro cajas de forma y tamaño muy similares y que contienen el mismo objeto, al aplicar NMS obtenemos la mejor caja, la que presenta más precisión, descartando el resto.
2. En segundo lugar el modelo usa una técnica llamada *hard negative mining*, que permite equilibrar las clases durante el proceso de entrenamiento. Con esta técnica sólo se utiliza un subconjunto de ejemplos negativos que presentan valores altos de coste durante el entrenamiento (por ejemplo falsos positivos) para cada iteración del entrenamiento. SSD deja una proporción de 3:1 de positivos con respecto a negativos.

El resto de detalles sobre este modelo se estudian en el apartado 2.2.1, ya que SSD Mobilenet es precisamente uno de los modelos con el que se trabaja en este proyecto.

Paralelamente a la línea de desarrollo con la API TensorFlow tenemos YOLO, *(Redmon y col., 2016)*, que se presenta como uno de los algoritmos más rápidos de detección actuales. Aunque hoy por hoy no presenta la mayor precisión en la detección, es una buena elección cuando se requiere de detección a tiempo real sin perder mucha precisión. Yolo ha lanzado las siguientes versiones:

- YOLO v1.
- YOLO v2. YOLO 9000.
- YOLO v3.

2.1.6. YOLO v1

YOLO es un sistema de detección de objetos cuyo objetivo es el procesado a tiempo real. La principal característica de YOLO es que utiliza una CNN para llevar a cabo la

detección de objetos y se distingue de otros modelos (salvo SSD), en que predice la caja y la clase detectada en una única red, en un solo paso (*Hui, 2018)*.

Como se describe en *Redmon y col. (2016),* para llevar a cabo la detección primero se divide la imagen en una cuadrícula de dimensiones ancho y alto *m*×*m*, figura 8, imagen de la izquierda). En cada una de las celdas se obtienen *N* posibles cajas de predicción y se calcula el nivel de confianza de cada una de ellas (imagen central), esto es, se obtienen *m*×*m*×*N* diferentes cajas, la mayor parte de ellas con un nivel de confianza muy bajo. Después de obtener estas predicciones se proceden a eliminar aquellas cajas situadas por debajo de un límite. A las cajas restantes se les aplica un paso de *non max suppression,* que elimina objetos que hayan podido ser detectados dos veces, obteniendo la predicción final, figura 8, imagen de la derecha

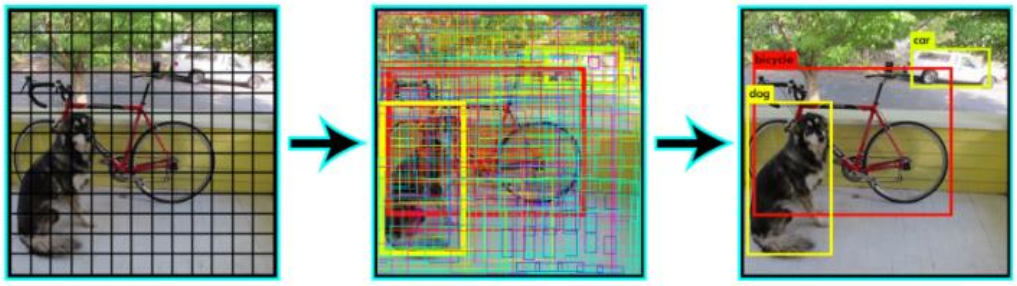


Figura 8. Obtención de predicción final en la detección con YOLO v1.
(Redmon y col. 2016)

Se proporcionan más detalles sobre YOLO v1 en el apartado 2.2.2, ya que es el otro modelo utilizado en el presente trabajo.

2.1.7. YOLO v2. YOLO 9000.

Como menciona *Hui (2018),* SSD se presenta como un gran competidor para YOLO proporcionando mejor precisión para procesado a tiempo real. Si lo comparamos con los detectores basados en región, YOLO tiene errores de localización mayores y la precisión relativa a la localización de objetos es menor. YOLOv2 es la segunda versión de YOLO y se desarrolla con el objetivo de mejorar la precisión de la versión anterior de forma significativa, haciéndola a su vez más rápida para poder competir con SSD.

Destacan las siguientes propiedades (*Hui, 2018):*

- Mejoras en la precisión.
  - Normalización de lotes en las capas de convolución.
  - Clasificador de alta resolución.
  - Convolución con anclado de cajas de predicción.
  - Clusters o grupos dimensionales.
  - Predicción directa de localización.
  - Características más detalladas (de grano fino).
  - Entrenamiento multi-escala.
- Mejoras en la velocidad

Como se recoge en *Redmon y Farhadi (2016),* YOLO 9000 extiende YOLO para detectar hasta 9000 clases usando clasificación jerárquica con un árbol de 9418 nodos. Combina muestras de COCO sobre 9000 clases de ImageNet. YOLO muestrea cuatro imágenes de ImageNet para cada imagen de COCO. Aprende a encontrar objetos utilizando los datos de detección de COCO y la clasificación con ImageNet.

2.1.8. YOLO v3.

YOLO v3 es la última versión de YOLO (*Redmon y col, 2018*). Como indica *Hui, (2018)*, mejora notablemente el rendimiento en la detección gracias fundamentalmente a las siguientes características:

- La función *softmax* se reemplaza con un clasificador logístico independiente que calcula la verosimilitud de una entrada con respecto a una determinada etiqueta. En vez de usar error cuadrático medio en el cálculo del coste en la clasificación, YOLO v3 usa entropía cruzada binaria para cada etiqueta. Esto hace que la complejidad computacional se vea reducida.
- Cambios en el modo de calcular la función de coste.
- YOLO v3 hace tres predicciones por localización. Cada predicción se compone de una caja de detección, un valor de objetividad y puntuación por clase.

2.2. Arquitectura de las redes seleccionadas.

En esta sección se resume la arquitectura de los modelos escogidos para llevar a cabo detección de objetos utilizando como entrada las imágenes que se reciben procedentes de la cámara del dron.

De entre los distintos modelos estudiados en el estado del arte, se escogen SSD y YOLO para el desarrollo del mismo, fundamentalmente por la similitud que presentan en sus arquitecturas y su *modus operandi*. El modelo de SSD que utilizamos es concretamente ssd_mobilenet_v1_coco, un modelo pre-entrenado en el conjunto de datos COCO (*TENSORFLOW MODEL ZOO,2018; COCO DATASET, 2018)*. En cuanto a YOLO, usamos la primera versión.

Este proyecto incluye un análisis comparativo de la velocidad de cada uno de los modelos estudiados en base a los resultados obtenidos en las pruebas realizadas. La motivación para desarrollar este análisis parte de la curiosidad en el estudio del estado del arte, donde se encuentran algunas comparaciones como la que se muestra en figura 9, en la que se compara la velocidad de los principales modelos recogidos en el apartado 2.1.

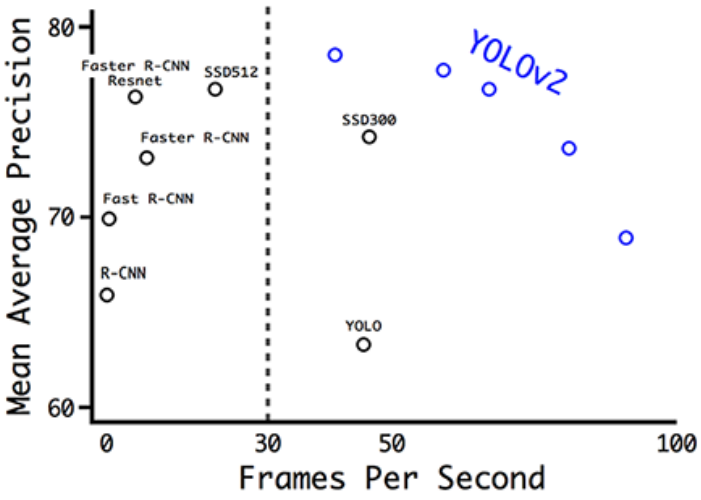


Figura 9. Comparativa de precisión y velocidad en distintos modelos para el conjunto de imágenes (dataset) VOC 2007. (*Redmon y Farhadi 2016)*

A continuación se describen brevemente las características y arquitecturas de las redes utilizadas en el presente proyecto.

2.2.1. Single-Shot Detector (SSD)

SSD se presenta como un método para detectar objetos en imágenes haciendo uso de una sola red neuronal profunda. Como indica *Forson (2017),* en su lanzamiento a finales de noviembre de 2016, SSD supuso un auténtico récord, llegando a una precisión promedio (mean Average Precision mAP) superior al 74 % a 59 frames por segundo (fps) en conjuntos de datos estándar como PascalVOC (*PASCAL VOC, 2018)* y COCO (*COCO DATASET,2018)*. Para comprender mejor SSD, a continuación se indica la procedencia del nombre de su arquitectura:

- **Single Shot:** Ambas tareas, clasificación y localización son llevados a cabo en un solo paso de propagación hacia delante de la red.
- **Multibox:** Este es el nombre de una técnica de regresión de cajas de detección desarrollada por *Szegedy y col. (2015)*.
- **Detector:** La red es un detector de objetos que también clasifica.

ARQUITECTURA DE RED

La figura 10 muestra el esquema de la arquitectura de red para la red SSD.

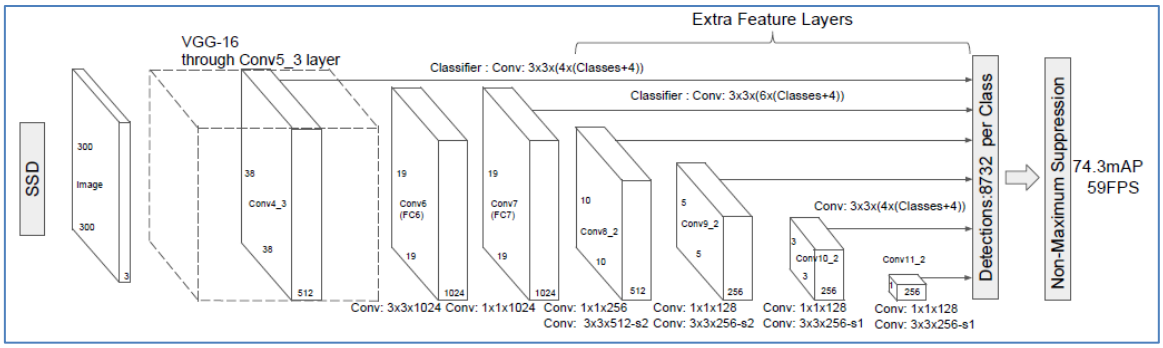


Figura 10. Arquitectura de red de SSD (*Liu y col. 2016).*

La arquitectura de red de SSD se basa en la arquitectura de VGG-16. La razón por la que se parte de la arquitectura de VGG es por su buen rendimiento y alta calidad en las tareas de clasificación de imágenes. Las capas totalmente conectadas son substituidas por un conjunto de redes convolucionales auxiliares (desde conv6 en adelante) que permiten la extracción de características en un re-escalado que reduce progresivamente el tamaño de las imágenes de entrada para la subsiguiente capa (*Liu y col (2016)).*

Como se menciona anteriormente, SSD se basa en la arquitectura Multibox, aunque incluye una serie de mejoras que vemos a lo largo de este apartado, que permiten que las tareas de clasificación y localización se hagan en un solo paso (Single Shot). A continuación se estudia el modelo Multibox.

Multibox

La técnica de regresión de cajas de predicción se inspira en el trabajo de *Szegedy y col. (2015),* un método de propuesta rápida de coordenadas de cajas de detección sin saber exactamente a qué clase pertenece (*class agnostic*). En la figura 11 se muestra un esquema de predicción convolucional multi-escala para el modelo Multibox.

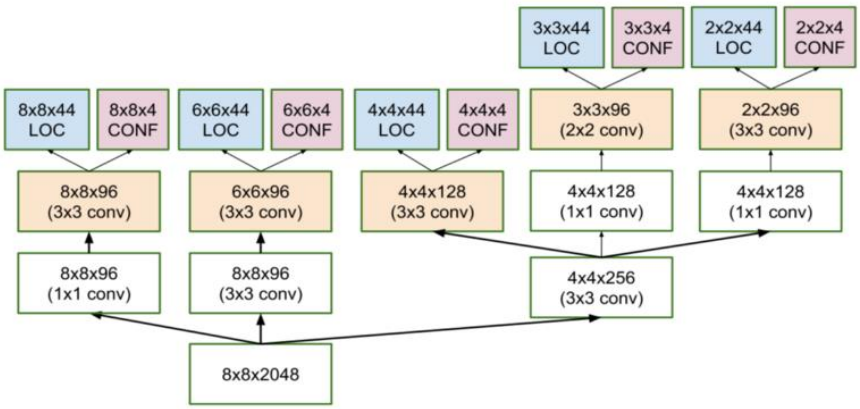


Figura 11. Esquema de predicción del modelo Multibox (*Szegedy y col., 2015)*

La función de coste en MultiBox también se combina con dos componentes críticos que hacen que se convierta en SSD:

- **Costes de Confianza.** Determina el grado de confianza de la objetividad en la computación de las cajas de detección (***confidence_loss, L_{conf})***

- **Costes de Localización.** Mide cuánto se aleja la predicción de las cajas de detección de la red con respecto a una imagen de referencia *(Ground Truth, Gt)* del conjunto de datos del entrenamiento **(*location loss,* L_{loc}).**

Según *Liu y col (2016)*, la expresión de coste, que mide lo lejos que estamos de una imagen de referencia *(Ground Truth, Gt)*, viene dada por la ecuación siguiente:

$$(x, c, l, g) = \frac{1}{N}\left(L_{conf}(x, c) + \alpha L_{loc}(x, l, g)\right) \qquad (2.1)$$

El término α contribuye a equilibrar la contribución de los costes en la localización, evitando que el error sea más grande en imágenes más grandes. Como es habitual en las técnicas de aprendizaje profundo, el objetivo es encontrar el parámetro que reduce la función de coste de forma más eficiente y que por tanto más nos aproxima la GT.

El resto de términos involucrados en la ecuación (2.1) son:

N: Número de cajas de detección coincidentes.

x: Toma el valor 1 cuando una caja de detección coincide con la imagen de referencia y cero si no coincide.

L: Predicción de los parámetros de la caja de detección.

g: Parámetros de la caja de detección de la imagen de referencia (GT)

c: Clase

IoU y Priors

En MultiBox (*Szegedy y col., 2015)*, los investigadores crean lo que denominaron *priores* (o *anchors* en terminología R-CNN), que son cajas de detección fijas pre calculadas que coinciden en gran medida con la distribución de las cajas de GT. En realidad, esos *priores* se seleccionan de tal forma que su *Ratio de Unión sobre Intersección,* IoU (Intersection over Union ratio) es superior a 0.5. Como se puede comprobar en la figura 12, un IoU de 0.5 puede que no sea suficientemente bueno, pero es un buen punto de inicio para el algoritmo de regresión de cajas de detección.

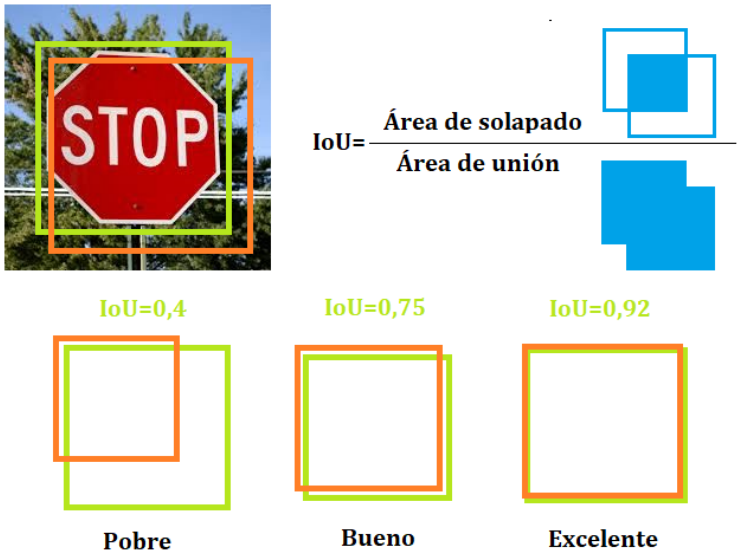


Figura 12. Ratio de Unión sobre Intersección

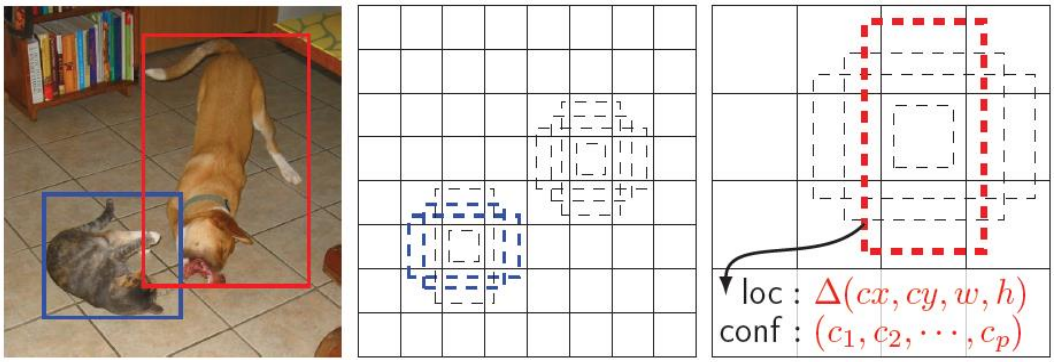
Mejoras de SSD

Volviendo a SSD, a partir de MultiBox se añadieron algunos matices para hacer a la red aún más capaz de localizar y clasificar objetos (*Forson, 2017)*.

Priores fijos.

A diferencia de MultiBox, cada mapa de características se asocia con un conjunto de cajas de detección de diferentes dimensiones y aspecto. Estos priores se seleccionan manualmente y con meticulosidad, mientras que los de MultiBox se seleccionaban en base al valor de su IoU, superior a 0,5. Esto hace que, de forma general, pueda generalizarse a cualquier tipo de entrada, sin que exista requisito de una fase de pre entrenamiento para la generación de priores. (*Forson, 2017)*.

Por ejemplo, asumiendo que hemos configurado dos puntos diagonalmente opuestos (x1, y1) y (x2, y2) para cada caja de detección predeterminada por cada celda ***b*** del mapa de características y que tenemos ***c*** clases para llevar a cabo la clasificación en un mapa de características dado de tamaño *f = m*n*, SSD debería computar *f*b*(4+c)* valores de características de este mapa de características, figura 13. (*Liu y col. 2016).*



(a) Imagen con cajas GT (b) Mapa de características 8x8 (c) Mapa de características 4x4

Figura 13. Cajas predeterminadas para mapas de características 8x8 y 4x4. (*Liu y col. 2016)*

Costes de localización:

SSD usa L1-Norm para calcular los costes de localización. Es efectivo y le proporciona más capacidad de maniobra, dado que no intenta ser perfecto píxel a píxel en la predicción de la caja de detección (*Forson, 2017)*.

Clasificación.

Multibox no lleva a cabo clasificación de objetos, mientras que SSD sí lo hace. Para cada caja de detección que se haya realizado predicción, se computan un conjunto de clases, para cada clase del conjunto de datos (*Forson, 2017)*.

ENTRENAMIENTO

Conjuntos de datos.

Es necesario disponer de conjuntos de datos con cajas de detección que indican la posición del objeto (GT) y con etiquetas de clases asignadas (una por caja de detección). Los conjuntos de datos Pascal VOC y COCO son un buen punto de partida.

Cajas de detección predeterminadas.

Se recomienda configurar un conjunto variado de cajas de detección predeterminada, de diferentes escalas y relación de aspecto para asegurar la detección de más objetos.

En (*Liu y col. 2016)* se menciona que son necesarias alrededor de seis cajas de detección por celda de mapa de características.

Mapas de características

Como menciona *Forson (2017),* los mapas de características son una representación de las características dominantes de la imagen a distintas escalas, por lo tanto, el hecho de aplicar MultiBox en múltiples mapas de características aumenta la probabilidad de cualquier objeto (pequeño o grande) para ser eventualmente detectado, localizado y clasificado de forma apropiada.

Minería de negativos

Durante el entrenamiento, la mayor parte de las cajas de detección tienen un bajo IoU y por lo tanto serán interpretadas como ejemplos de entrenamiento negativos. Debido a esto acabaríamos con un número desproporcionado de ejemplos negativos en nuestro conjunto de imágenes de entrenamiento. Por tanto, en lugar de usar todas las predicciones negativas, se aconseja mantener una relación de negativos con respecto a positivos de 3 a 1. La razón por la que es necesario mantener muestras negativas es porque la red también necesita aprender cuándo está incurriendo en un error (*Liu y col (2016)).*

Aumento de datos

En *Liu y col. 2016)]* apuntan que el aumento de datos, como otras aplicaciones de Deep Learning, han sido cruciales para entrenar la red de cara a obtener más robustez ante entradas de varios tamaños. Además añaden ejemplos de entrenamiento con parches sobre la imagen original en diferentes ratios IoU. Por otra parte, cada imagen también cambia de forma especular horizontalmente con una confianza de 0,5, haciendo, por tanto, que los objetos potenciales que aparezcan por la derecha o por la izquierda tengan una probabilidad de detección similar.

Non-Maximum Supression (NMS)

Dado un gran número de cajas generadas durante un paso de una entrada por SSD, es esencial depurar la mayor parte de las cajas de detección gracias a la aplicación de una técnica conocida como *non-maximum suppression*: Las cajas con umbral de costes

de confianza pequeño (por ejemplo 0.01) y un IoU inferior a otro cierto umbral (por ejemplo 0,45) se descartan, quedándonos únicamente con las mejores predicciones. Esto asegura que sólo las mejores predicciones son reentrenadas por la red, mientras que las que presentan más ruido son eliminadas. La imagen de la figura 23 muestra un esquema de este proceso (*Liu y col. 2016).*

Figura 14. Filtrado de cajas de detección gracias a non-maximum suppression (*Liu y col. 2016)*

2.2.2. You only Look Once (YOLO)

YOLO plantea la detección de objetos como un problema de regresión, directamente desde los pixeles de una imagen hasta las coordenadas de la caja delimitadora y la confianza de detección de clases. El nombre YOLO, viene precisamente de este hecho, ya que sólo es necesario mirar una vez para detectar qué objetos están presentes y qué son.

YOLO se presenta como una única red convolucional que simultáneamente predice la posición con cajas de detección múltiples e indica la confianza de la clase detectada en esa caja. En *Redmon y col. (2016) se menciona que el sistema* funciona a 45 fps sin procesamiento por lotes en una GPU Titan X.

ARQUITECTURA DE RED

YOLO se implementa como una red convolucional y se evalúa en el conjunto de datos PASCAL VOC (*2018)*. Las capas iniciales de la red extraen características de la imagen mientras las capas completamente conectadas predicen la confianza en la detección y las coordenadas de las salidas.

La arquitectura de red se basa en el modelo GoogLeNet (*Szegedy y col., 2014)* para clasificación de imágenes. La red tiene 24 capas convolucionales seguidas de 2 capas completamente conectadas. En lugar de los módulos de incepción usados por GoogLeNet, simplemente se usa una capa de reducción de 1x1 seguida de una capa convolucional de 3x3. La arquitectura de red completa se muestra en la figura 15.

La salida final de la red es un tensor de predicciones de dimensiones 7× 7×30.

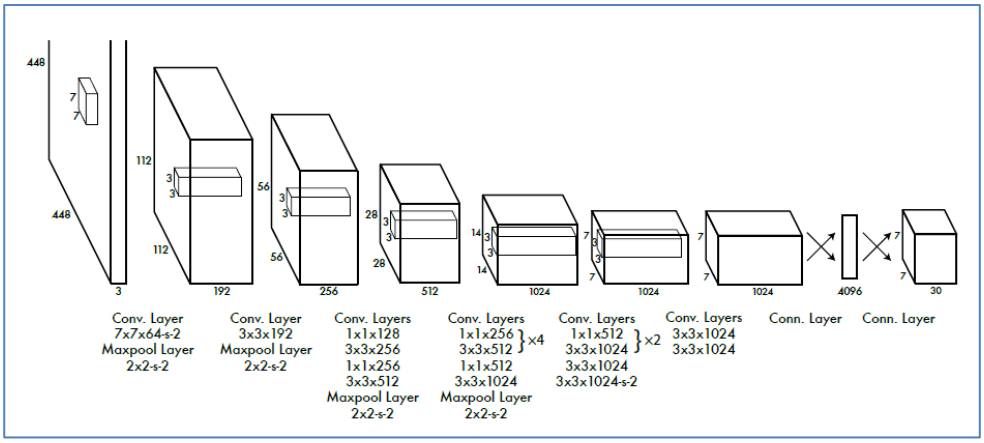


Figura 15. Arquitectura de red de YOLO. *Redmon y col. (2016)*

FUNCIÓN DE COSTE

La función de coste *Redmon y col. (2016)* es realmente extensa, así que la estudiamos en tres partes *Redmon y col. (2016)*:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \qquad (2.2)$$

Esta ecuación computa los costes referidos a la posición (x, y) de la predicción de la caja de detección. La función realiza una suma sobre cada predicción de caja de

detección (j=0…B) de cada celda (i=0…S^2). $\mathbb{1}$ se define como se indica a continuación:

- 1 si un objeto está presente en la celda *i* y la *j*-ésima predicción de caja de detección es "responsable" de esa predicción.
- 0 si no es así.

Sabemos qué predicción es responsable para un objeto basándonos en su inserción sobre la unión (IoU). La predicción con el IoU más alto con respecto GT será la responsable.

Con respecto al resto de términos, *(x, y)* es la posición de la caja de detección y $(\hat{x}, \hat{y})$ es la posición actual de los datos del entrenamiento.

Respecto de la segunda parte:

$$\boldsymbol{\lambda_{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[\left(\sqrt{\boldsymbol{w_i}} - \sqrt{\widehat{\boldsymbol{w}}_i} \right)^2 + \left(\sqrt{\boldsymbol{h_i}} - \sqrt{\widehat{\boldsymbol{h}}_\iota} \right)^2 \right] \qquad (2.3)$$

Este es el coste relativo al ancho/alto de la caja de predicción. El error métrico debería reflejar que las pequeñas desviaciones en las cajas más grandes importan menos que en las cajas más pequeñas. Para abordar esto se predicen las raíces cuadradas del ancho y alto de la caja de predicción en lugar de hacerlo de forma directa. En la ecuación [2.3], (w,h) corresponde al ancho y alto de la caja de detección, mientras que $(\hat{w}, \hat{h})$ corresponde a las dimensiones de los datos de entrenamiento.

Respecto de la tercera parte:

$$\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left(\boldsymbol{C_i} - \widehat{\boldsymbol{C}}_i \right)^2 + \boldsymbol{\lambda_{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} \left(\boldsymbol{C_i} - \widehat{\boldsymbol{C}}_i \right)^2 \qquad (2.4)$$

En esta ecuación computamos los costes asociados a la puntuación de confianza para cada predicción de caja de detección. $\boldsymbol{C}$ es la puntuación d la confianza y $\hat{C}$ es la intersección sobre la unión (IoU) de la predicción de la caja de detección con la *GT*. $\mathbb{1}^{\mathbf{obj}}$ es igual a 1 cuando hay un objeto en la celda y 0 si no lo hay. $\mathbb{1}^{\mathbf{noobj}}$ es lo contrario.

El parámetro λ que aparece en las ecuaciones se usa para compensar de forma diferente las distintas partes de las funciones de coste. Es necesario para proporcionar estabilidad al sistema. Se penaliza más la parte de las coordenadas de predicción (λ_{coord}=5) y menos la parte correspondiente a la confianza en las predicciones cuando no hay objeto presente (λ_{noobj}=0.5)

La última parte de la función de coste es la parte correspondiente a la clasificación:

$$\sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \qquad (2.5)$$

Esta última ecuación tiene un aspecto similar al error de sumas de cuadrados normal para clasificación, excepto por el término $\mathbb{1}_i^{obj}$. Este término se usa porque de esta forma no penalizamos el error de clasificación cuando no hay objeto presente en la celda. $\hat{p}_i(c)$ denota la confianza de clasificación de una clase c en la celda i. Si un objeto es detectado, el coste de clasificación en cada celda es el error cuadrático de la confianza condicional por cada clase.

3. Capítulo 3. Integración del sistema.

Como se describe en el Capítulo 1, el objetivo de este TFM es llevar a cabo detección de objetos mediante técnicas de aprendizaje profundo con las imágenes recibidas de un dron mientras éste se pilota de forma remota. Para llegar a conseguir este objetivo se ha seguido una evolución que comienza con unas primeras pruebas en las que se hizo uso de máquinas virtuales, en las que se obtiene resultados pobres en lo que respecta a velocidad de proceso y reacción del sistema, hasta una integración final haciendo uso de hardware con características adecuadas que, como se verá en la presentación de resultados, nos permiten un notable aumento de velocidad y también una mejora en la detección, acercándonos a tiempo real, con las limitaciones tanto de hardware como de software que presenta el sistema integrado.

A continuación se describe tanto el punto de partida del proyecto como la integración final usada para llevar a cabo las pruebas, sin entrar en los resultados obtenidos, que estudiaremos con detenimiento en el capítulo 5.

3.1. Descripción del software base utilizado.

Este apartado describe de forma breve el principal software utilizado para el desarrollo de este proyecto.

En el planteamiento inicial se dispone del dron AR2Drone de Parrot y de un ordenador portátil. Tras la ejecución de las pruebas del primer planteamiento se descubren notables limitaciones en el procesado de imágenes achacables al hardware utilizado. Así pues, se decide un replanteamiento del proyecto, introduciendo una serie de dispositivos que permiten mejorar considerablemente la capacidad de procesado del sistema, como se verá a lo largo de los siguientes apartados.

3.1.1. Robotic Operating System (ROS)

El framework ROS (Robotic Operating System) (*ROS, 2018)* es clave para el desarrollo de este trabajo dado que en todo momento utilizaremos este sistema para la transferencia y procesado de imágenes.

ROS es un framework flexible para desarrollar software destinado a la robótica. Es una colección de herramientas, librerías y convenciones cuyo propósito es simplificar la tarea de crear comportamientos complejos y robustos de robots a través de una gran variedad de plataformas de robótica.

Como se comenta en *ROS,* nace con el propósito de desarrollar software enfocado a la robótica de forma colaborativa. Por ejemplo, un laboratorio puede tener expertos en el mapeo de ambiente en exteriores y por tanto puede contribuir a un sistema de clases mundial para la creación de mapas. Otro grupo puede tener expertos en el uso de esos mapas para navegar y otro grupo puede haber encontrado una solución basada en visión por computador que funciona bien para reconocer pequeños objetos basado en clustering. ROS se diseñó específicamente para grupos como éstos, que quieran colaborar y construir a partir del trabajo desarrollado de otros grupos. El sistema ROS cuenta hoy en día con más de 10000 usuarios a lo largo de todo el mundo, trabajando desde proyectos por simple afición hasta grandes proyectos destinados a la automatización industrial.

A continuación se describe de forma general el funcionamiento de ROS a alto nivel.

ROS se inicia al ejecutarse el ROS Master. El Master permite a otros elementos del entorno ROS encontrarse y comunicarse entre sí. De este modo, no tenemos que estar especificando constantemente, por ejemplo, el envío de las lecturas de un sensor a un ordenador en 127.0.0.1. Simplemente le diremos al nodo 1 que mande mensajes al nodo 2. La figura 16 muestra el esquema general de ROS.

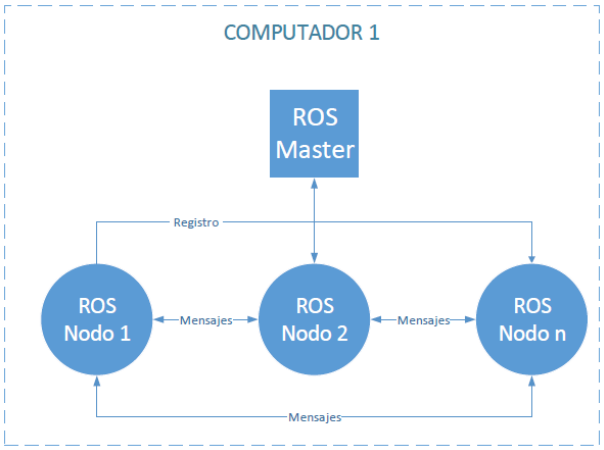


Figura 16. Interacción de nodos en un sistema ROS

Pongamos, como ejemplo ilustrativo, que tenemos una cámara en nuestro robot cuyas imágenes queremos procesar directamente en el mismo En el ejemplo esquematizado en la figura 17 tenemos un nodo de cámara que controla la comunicación de la cámara y un nodo de procesado de imágenes en el robot que procesa los datos de imagen. Para empezar, todos los nodos se han registrado con el Master. Pensaremos en el Master como una tabla de seguimiento donde todos los nodos van para encontrar exactamente dónde enviar los mensajes.

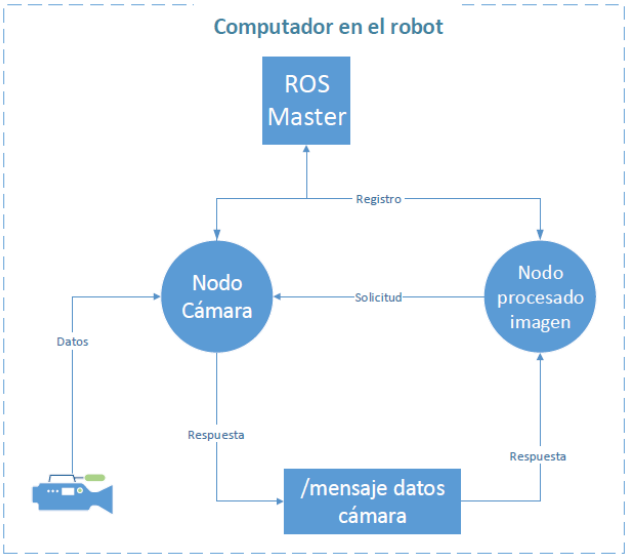


Figura 17. Publicación y suscripción de tópicos ROS.

En el proceso de registro con el Máster, el estado del nodo de la cámara publicará un tópico llamado /mensaje datos cámara (por ejemplo). El nodo de procesado de imagen estará suscrito a ese tópico

Por lo tanto, una vez el nodo de la cámara recibe algún dato de ésta, dicho nodo lo envía a través del tópico /mensaje datos directamente hacia el otro nodo (fundamentalmente esta comunicación tiene lugar a través de protocolo TCP/IP).

También existe la posibilidad de solicitar imágenes a la cámara en un momento específico. Para hacer esto, ROS implementa los denominados Servicios, aunque en este proyecto no se abordan, trabajamos únicamente con nodos y tópicos.

3.1.2. OpenCV

OpenCV (*2018)* es una librería multiplataforma de código abierto que nos proporciona de bloques fundamentales para experimentos de visión por computador y aprendizaje automático. Ofrece interfaces de alto nivel para capturar, procesar y presentar datos de imagen.

Open CV se usa de forma amplia tanto en el ámbito académico como industrial. Hoy en día, la visión por computador puede llegar al consumidor en muchos contextos distintos tales como videocámaras, cámaras de teléfonos móviles, sensores de videojuegos entre otros.

OpenCV fue creado para ofrecer una infraestructura común de visión por computador y para acelerar el uso de la percepción de máquinas en productos comerciales.

Como se menciona en *Solem (2012),* OpenCV en entorno Python, que es como utiliza a lo largo de este proyecto, nos ayuda a explorar soluciones en un lenguaje de alto nivel en un formato estandarizado que nos permite inter-operar con librerías científicas tales como NumPy (2018).

OpenCV ofrece más de 2500 algoritmos optimizados, que incluyen un exhaustivo estado del arte de visión por computador y aprendizaje automático. Estos algoritmos pueden ser usados para detectar y reconocer caras, identificar objetos, clasificar acciones humanas en videos, seguir movimientos de cámara, seguir objetos, extraer modelos 3D de objetos, producir nubes de puntos 3D, etc.

3.1.3. TensorFlow.

TensorFlow (*2018)* es una biblioteca de software libre que nos permite realizar cálculos numéricos mediante diagramas de flujo de datos. En su origen TensorFlow fue el fruto del trabajo de investigadores e ingenieros de Google Brain Team, que formaban parte del departamento de investigación de aprendizaje automático de Google.

TensorFlow es una interfaz que permite expresar e implementar algoritmos de aprendizaje automático. La computación expresada usando TensorFLow puede ser ejecutada sin ningún cambio o simplemente pequeñas modificaciones en un heterogéneo rango de dispositivos que van desde dispositivos móviles como teléfonos o tablets hasta grandes dispositivos distribuidos en cientos de máquinas computacionales como GPUs. El sistema es flexible y puede usarse para implementar una gran variedad de métodos, procesos, algoritmos, incluyendo entrenamiento y algoritmos de inferencia para modelos de redes neuronales. Ha sido ampliamente utilizado como soporte para investigación y para el desarrollo de sistemas de aprendizaje automático hasta su fase de producción para docenas de áreas de ciencias de la computación y otros campos, incluyendo el reconocimiento de voz, visión por computador, robótica, recuperación de información, procesado natural de idiomas, extracción de información geográfica y medicina computacional.

En el contexto del presente trabajo, un tensor es una entidad algebraica que es utilizada en la inteligencia artificial para crear redes neuronales y mejorar las capacidades del aprendizaje automático. En la figura 18 se muestra parte del grafo de un modelo generado gracias a TensorFlow. Entre otras, podemos ver la capa *softmax*, que es una de las de referencia.

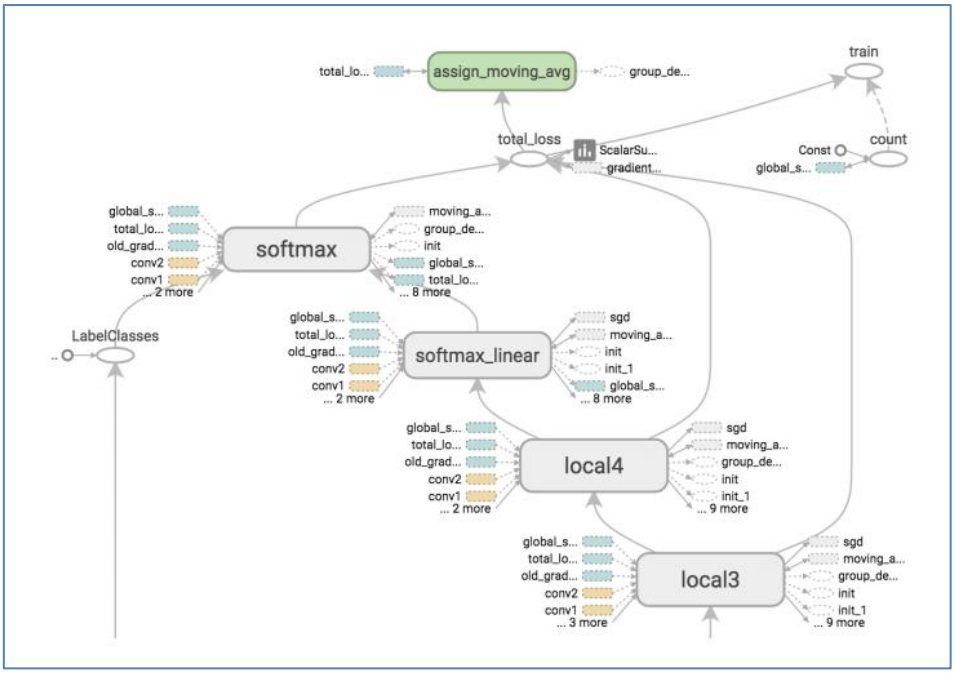


Figura 18. Gráfico de visualización de modelo de red neuronal convolucional.

TENSORFLOW (2018)

3.1.4. CUDA. (Computer Unified Device Architecture)

CUDA ® (2018) es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema.

Gracias al número de desarrolladores, investigadores y científicos trabajando sobre estos sistemas, se están encontrando innumerables aplicaciones prácticas para esta tecnología en campos tales como el procesado de video e imágenes, biología y química computacional, simulación de la dinámica de fluidos, análisis sísmico, etc.

Los sistemas informáticos están pasando de realizar el "procesamiento central" en la CPU a realizar "co-procesamiento" repartido entre la CPU y la GPU. Para posibilitar este nuevo paradigma computacional, NVIDIA ha desarrollado la arquitectura de cálculo paralelo de *CUDA*.

3.2. Integración del sistema.

Este apartado tiene como propósito describir los pasos llevados a cabo para integrar el sistema. Se parte de un primer planteamiento en el que se trabaja con máquinas virtuales. Como se comenta al principio de este capítulo, vistos los resultados obtenidos con el sistema planteado, se opta por evolucionar a un nuevo planteamiento que mantiene el uso de una de las máquinas virtuales y sustituye la segunda por el sistema embebido *JETSON TX2 (2018)* de *Nvidia*, con el objetivo de optimizar el proceso de detección y acercarnos a la detección a tiempo real. A continuación vemos los detalles de la integración de cada uno de los sistemas con los que se experimenta.

3.2.1. Planteamiento 1. Máquinas virtuales

El planteamiento inicial de este proyecto tenía ciertas diferencias con el trabajo finalmente desarrollado. La idea inicial era que el dron llevara a cabo vuelo autónomo, partiendo del driver *ARDRONE_AUTONOMY (2018)* y del driver experimental *ARDRONE AUTONOMY GPS (2018)* mediante los datos obtenidos de un dispositivo GPS añadido al dron.

Tras llevar a cabo ciertas pruebas con el sistema, conseguir leer datos GPS provenientes del dron y probar el vuelo autónomo, se decide abandonar esta idea por el riesgo de perder el dron durante los tests, dado que no se obtiene el control esperado, posiblemente porque el driver en el que nos basamos es experimental.

Se realiza un segundo planteamiento en el que se pretende diseñar un joystick de control en una interfaz de usuario de MATLAB *(2018)*, basada en un trabajo anterior de prácticas del máster. Después de lograr algunos resultados y tras una serie de pruebas, se decide abandonar esta idea también, debido a la lenta reacción del joystick diseñado y el escaso control sobre el dron.

Se opta por una tercera opción basada en el trabajo recogido en *Levy (2014, 2018)*. El driver creado por el desarrollador permite el control sobre el dron a partir de un mando de PlayStation, que es por el que optamos. La figura 19 muestra el esquema básico para el primer planteamiento.

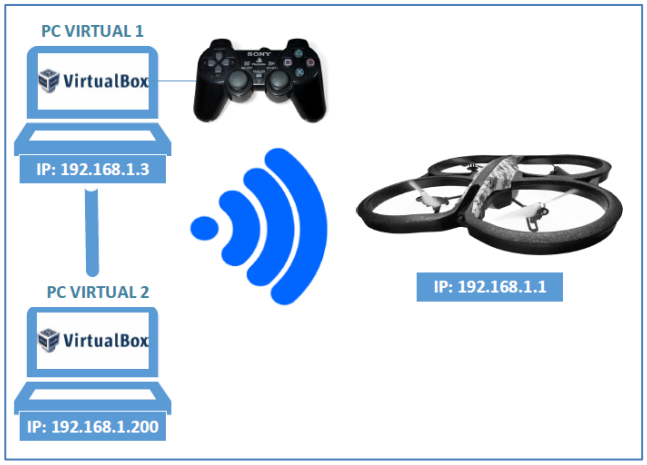


Figura 19. Esquema básico planteamiento 1

Como se ve en la figura 19, trabajamos con dos máquinas virtuales *VIRTUAL BOX (2018)* corriendo sobre entorno Windows 10. El PC Virtual 1 va a tener como cometido fundamental controlar el dron con el driver mencionado y tomar, asimismo, las imágenes procedentes del mismo. Esa imagen se encapsulará en un tópico ROS. Desde el PC Virtual 2 se iniciará un nuevo nodo ROS que se subscribirá al tópico que contiene la imagen que procede del dron para generar un nuevo tópico con la imagen ya procesada gracias a la red neuronal. El esquema de la red ROS creada se muestra en la figura 20.

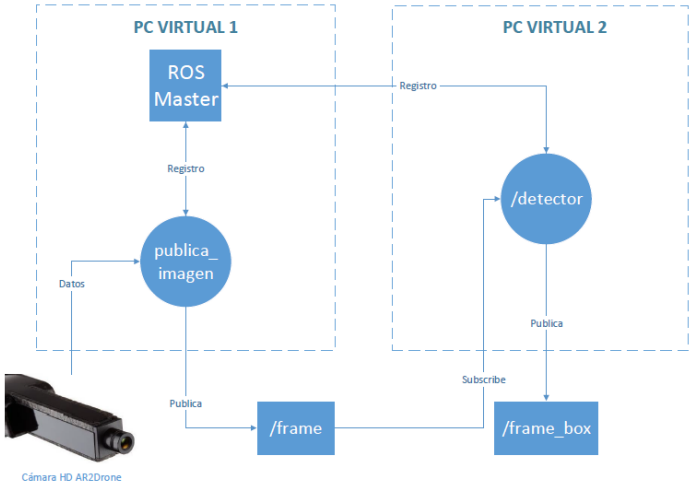


Figura 20. Esquema ROS planteamiento 1

Una vez definido el sistema comienza a instalarse el software necesario. El proceso de instalación resulta tedioso y complicado cuando se parte de escasa experiencia, es por esto que se opta por trabajar con *máquina virtual*. De este modo existe la posibilidad de hacer copias de seguridad en ciertos momentos críticos de la instalación, que nos permite volver atrás en caso de incurrir en algún error.

Es de vital importancia trabajar con las versiones correctas del software instalado e integrado en el sistema. Debemos, por tanto, poner una atención especial para instalar siempre software y librerías compatibles con nuestro sistema.

A continuación se enumera el software necesario en ambas máquinas virtuales, esquematizado en la figura 21:

- VIRTUALBOX (2018) sobre Windows 10.
- Ubuntu 16.04 (XENIAL XERUS, 2018).
- ROS Kinetic (2018).
- Python 2.7 (*Solem, 2012)*
- Open CV (2018).
- NumPy (2018).
- CV_bridge. Es una librería extra de ROS llamada *ROS CV_BRIDGE (2018)*, que permite transformar imágenes de formato OpenCV a mensajes ROS de tipo imagen y viceversa.

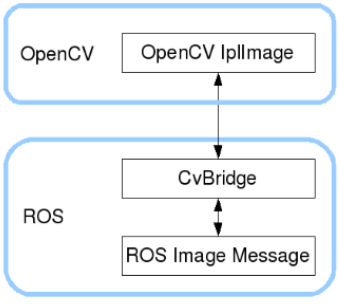


Figura 21. Transformación de imágenes con cv_bridge. *ROS CV_BRIDGE (2018)*

Se instala OpenCV siguiendo los pasos indicados en *INSTALACIÓN OPENCV (2018).* Todo el software indicado anteriormente se instala en ambas máquinas virtuales. Además de este, en el PC Virtual 2 se instala TensorFlow siguiendo los pasos indicados en *INSTALACIÓN TENSORFLOW (2018).*

Una vez instalado el software básico, se lleva a cabo la configuración de red TCP/IP, así como la configuración de red ROS. Como se puede comprobar en la figura 22, el ROS Master se configura para funcionar sobre el PC Virtual 1. De esta forma, desde el PC virtual 2 podrán visualizarse tanto los nodos como los tópicos activos en ROS.

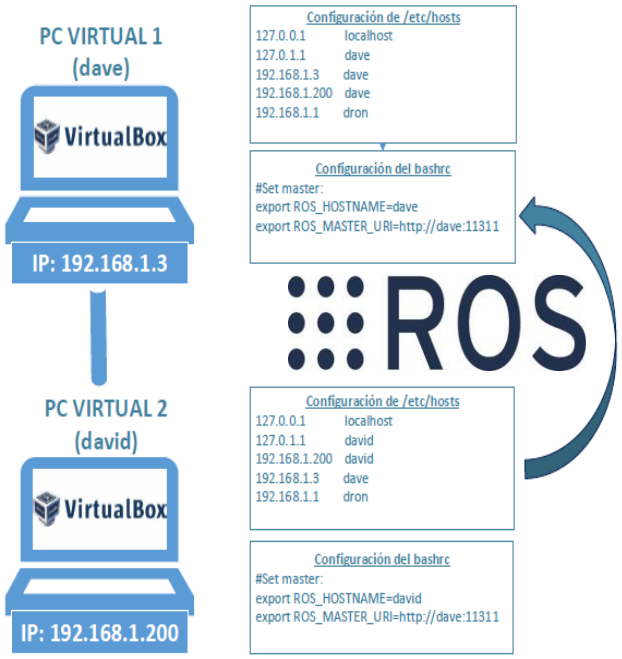


Figura 22. Configuración de red en Ubuntu y ROS.

El siguiente paso consiste en preparar el entorno de ambas máquinas virtuales, tal y como recoge el Anexo II. De esta forma tendremos el sistema preparado para lanzar la red neuronal y comenzar a detectar objetos.

El script planteado inicialmente para llevar a cabo el reconocimiento de imágenes desde el PC Virtual 2 está desarrollado en Python y basado en TensorFlow, OpenCV y ROS como base para la transmisión de datos. La detección de objetos se basa en el modelo ssd_mobilenet_v1_coco, que es, como se menciona en el apartado 2.2, un modelo pre-entrenado en el conjunto de datos COCO *(TENSORFLOW MODEL ZOO (2018); COCO DATASET (2018)).*

Desde el PC Virtual 2 comprobamos que recibimos correctamente el tópico */frame*, aunque es evidente que el ratio de imágenes recibidos en esta máquina virtual es significativamente inferior a los fps del mismo tópico que obtenemos en el PC Virtual 1.

Al ejecutar la red neuronal en el PC Virtual 2 descubrimos que el ratio de imágenes ya procesadas (publicadas en el tópico */frame_box*) es en todo caso inferior a 3 fps. Los datos obtenidos se estudiarán más en detalle en el siguiente capítulo de este trabajo.

Dada la baja capacidad de procesado de imágenes con el método propuesto y en base al estado del arte, en el que veíamos que, haciendo uso del hardware adecuado, la velocidad debería ser mucho mayor (figura), se comienzan a investigar posibles métodos para mejorar el proceso.

La primera decisión que se toma es sustituir la Máquina Virtual 2 por una GPU, la cual a priori debería mejorar notablemente las características de hardware de la máquina en que se ejecuta la red neuronal.

En el siguiente apartado se muestra la integración del sistema con el nuevo dispositivo añadido al sistema.

3.2.2. Planteamiento 2. Procesado con GPU y TensorFlow.

Siguiendo el mismo planteamiento que en el apartado anterior, en el PC Virtual 1 tendremos el subsistema *ARDroneAutoPylot, Levy (2014, 2018),* el cual nos permite controlar el dron a través del joystick y obtener las imágenes de su cámara en el tópico ROS /frame.

Introducimos el sistema embebido JETSON TX2 (*2018)*, con las características recogidas en el Anexo I para tratar de mejorar la velocidad de proceso que obteníamos en el planteamiento 1. Lo ponemos en el lugar del PC Virtual 2 , de esta forma, será en la GPU donde se ejecute la red neuronal creada por TensorFlow.

Para añadir la GPU al el sistema, contamos, asimismo con un *switch* y un punto de acceso Wi-Fi. De esta forma y gracias a la configuración de red que podemos ver en la figura 23, el sistema queda integrado. Al igual que sucedía en el primer planteamiento, configuramos el PC Virtual 1 como Master, por lo tanto, es en esa máquina donde debemos ejecutar el nodo máster mediante el comando *roscore*.

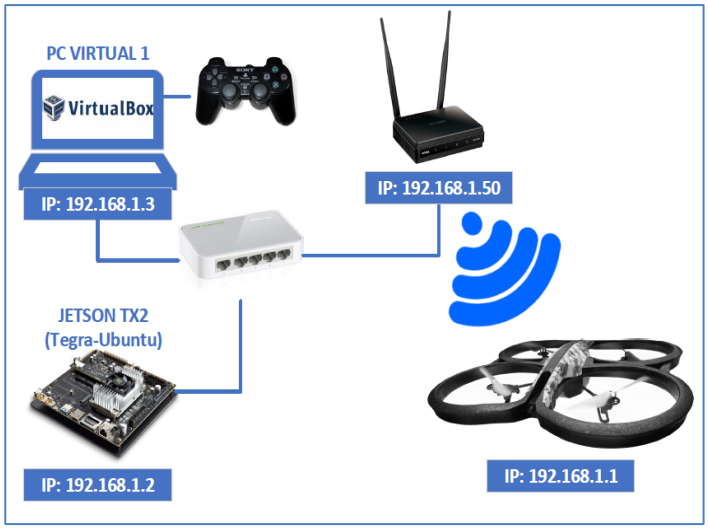


Figura 23. Esquema básico de red planteamiento 2.

No hay cambios importantes en lo que respecta a la configuración del PC Virtual 1. Sin embargo, sí que existen numerosas diferencias en la forma de instalar software y configurar la Jetson TX2. El Anexo III recoge un pequeño resumen de los principales pasos a tener en cuenta a la hora de instalar el software necesario. Una vez más se hace fundamental tener un gran control sobre las versiones que utilizamos, ya que, algo que parece sencillo a priori, puede convertirse en un tedioso proceso, pudiendo arrastrar errores derivados de incompatibilidades en las distintas versiones del software y librerías implicadas.

El planteamiento en lo que se refiere a la estructura ROS es el mismo que en planteamiento 1, reflejado en la figura 20, sólo cambian los nombres de las máquinas implicadas.

Tras llevar a cabo las primeras pruebas con este planteamiento, y tal como podemos comprobar en los resultados recogidos en el capítulo 4, continuamos teniendo resultados muy pobres en lo que respecta al número de fps obtenidos en la imagen ya procesada por el detector. Observamos asimismo un retardo de entre tres y cuatro segundos en la recepción de la imagen ya procesada. Comprobamos también que existe un gran retardo en la recepción de imágenes desde el nodo /frame en la GPU, lo cual a priori resulta extraño, dado que se esperaba una notable mejoría en ese sentido al introducir la Jetson TX2 en el sistema. Esto nos lleva a sospechar que el cuello de botella se encuentra en el transporte de la imagen desde el PC Virtual 1 hasta la Jetson. Para verificar que esa es la razón por la que el número de fps

desciende del modo en el que lo hace en el envío de las imágenes provenientes del tópico /frame desde el PC Virtual 1 a la GPU, llevamos a cabo un pequeño cambio sobre el script autopylot_agent.py con el fin de reducir el tamaño de la imagen transportada

```
# Crear imagen a todo color desde bytes
image_cv = np.frombuffer(img_bytes, np.uint8)
image_cv = np.ndarray.reshape(image_cv, (img_height, img_width, 3))
resize_cv = cv2.resize(image_cv, dsize(160,90), interpolation=cv2.INTER_CUBIC
image_ros = bridge.cv2_to_imgmsg(resize_cv)
pub.publish(image_ros)
```

Una vez realizado el cambio y compilado de nuevo el sistema, volvemos a lanzar ardrone_autopylot, descubriendo que hay una gran mejora en el transporte de la imagen. Comprobamos un notable aumento en el número de *fps* del tópico /frame recibido en la Jetson. La única incidencia que se encuentra es que la imagen procesada se ve reducida enormemente y se hace complicado visualizar la detección del sistema de forma correcta. Una vez verificado que el cuello de botella está en el transporte del tópico /frame (conviene recordar que la imagen procedente de la cámara del dron es HD (720p), se sigue buscando un método que mejore el redimensionamiento de la imagen.

La opción por la que se opta finalmente es la compresión de la imagen gracias al paquete ROS *image_transport (2018)*. Se hace uso de la opción que el sistema tiene para Python por la sencillez que presenta, sin necesidad de modificar código, simplemente lanzando nuevos tópicos que permiten la compresión y transporte de la imagen; concretamente se trata de realizar una suscripción al tópico */frame*, republicarlo en formato comprimido desde el PC Virtual 1 y volverlo a republicar en formato descomprimido, ya en la Jetson. Los comandos que utilizamos para llevar a cabo esta operación se relacionan a continuación.

- Desde el PC Virtual 1 ejecutamos

```
rosrun image_transport republish raw in:=/frame compressed out:=/frame_repub
```

De esta forma creamos un nuevo tópico llamado */frame_repub* que contiene las imágenes comprimidas procedentes de la cámara del dron.

- Desde el terminal de la Jetson ejecutamos el siguiente comando:

```
rosrun image_transport republish compressed in:=/frame_repub raw
out:=/frame_descomprimido
```

De esta forma, obtenemos las imágenes procedentes del dron en formato HD (720p) en la el sistema embebido Jetson TX2 con una frecuencia de 30 fps. Por tanto, el método encontrado para el transporte, es sencillo y eficiente. De esta forma estamos listos para proseguir optimizando el rendimiento de nuestro sistema para aproximarnos lo más posible al tiempo real.

El esquema de la figura 24 recoge el esquema ROS utilizado para transportar la imagen procedente del dron en alta definición.

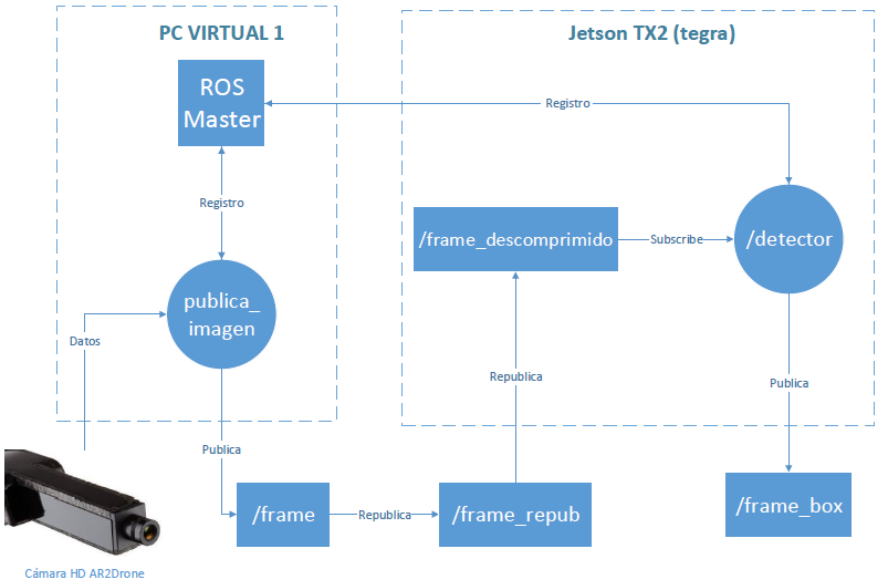


Figura 24. Esquema ROS del planteamiento 2 con compresión de imagen.

Una vez integrado el sistema y publicados los tópicos como se indica en la figura 24, se procede al lanzamiento del script de reconocimiento de objetos desde la Jetson TX2. Encontramos dos errores en la ejecución del script original.

- No se encuentra la GPU. El sistema funciona, pero se comprueba que no se está usando la GPU. Esto hace que la velocidad de proceso baje considerablemente. Monitorizamos el rendimiento del hardware gracias al

comando *sudo ./tegrastats | grep GR3D*, resaltando así el porcentaje de uso de la GPU.

- En algunas ocasiones parece que se llega a saturar la memoria RAM del sistema embebido y no llega a construirse el grafo de TensorFlow.

Tras investigar ambos errores y llevar a cabo pruebas con varias soluciones, se comprueba que las siguientes líneas de código consiguen resolver ambos problemas.

```
os.environ["CUDA_VISIBLE_DEVICES"] = '0' #usar GPU con ID=0
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.6 # Reservar máximo 60%
de la memoria RAM
config.gpu_options.allow_growth = True #reservar dinámicamente
```

Una vez resueltas estas incidencias, se consigue lanzar adecuadamente el script, comprobando que el número de *fps* en el tópico */frame_box* ha mejorado notablemente.

Para optimizar aún más el rendimiento de la GPU, se hace uso de uno de los modos que presenta la Jetson TX2, concretamente se ejecuta el modo Max-N gracias al comando *sudo nvpmodel –m 0*. Podemos monitorizar que el modo de funcionamiento es el que queremos, ejecutando el comando *sudo nvpmodel –q --verbose.* La tabla 1 muestra los distintos modos de funcionamiento con los que podemos trabajar con la Jetson TX2.

Modo	Nombre Modo	Denver 2	Frecuencia (GHz)	ARM A57	Frecuencia (GHz)	Frecuencia GPU (GHz)
0	Max-N	2	2.0	4	2.0	1.30
1	Max-Q	0		4	1.2	0.85
2	Max-P Core-All	2	1.4	4	1.4	1.12
3	Max-P ARM	0		4	2.0	1.12
4	Max-P Denver	2	2.0	0		1.12

Tabla 1. Modos de funcionamiento JETSON TX2. *NVPMODEL JETSON (2018)*

Llevamos a cabo otro paso para optimizar el uso del hardware, aumentamos la velocidad del reloj tanto de en CPU como en GPU. Gracias al comando *sudo ./jetson_clocks.sh* apagaremos el modo ahorro de energía y el sistema estará a su máximo rendimiento, veremos, asimismo, cómo arranca el ventilador de la Jetson TX2.

Estos últimos pasos nos permiten optimizar el hardware, mejorando la velocidad de proceso y acercarnos a procesado en tiempo real. En el capítulo 4 de este trabajo podremos comprobar los resultados obtenidos con el sistema ya optimizado.

Una vez conseguida esta optimización se plantean dos casos de detección, el primero, que es el con el que se inicia este proyecto y con el que se ha trabajado para conseguir la optimización, detección de objetos basada en TensorFlow y SSD Mobilenet, concretamente el modelo ssd_mobilentet_v1_coco y un segundo caso, detección de objetos basada en un modelo reentrenado para detectar señales de tráfico (stop y ceda el paso).

3.2.2.1. Detección de objetos basada en ssd_mobilenet_v1_coco.

Hasta este punto hemos estado trabajando en todo momento con el modelo *ssd_mobilenet_v1_coco* (*TENSORFLOW MODEL ZOO, 2018)*, modelo procedente del modelo zoo y que nos permite detectar hasta 90 tipos de clases

En el script que se utiliza para la detección con el modelo original se asigna un factor de 0,4 en la puntuación que se debe obtener en cada detección, por tanto, el sistema sólo nos presentará como positivos y por tanto marcará con cajas aquellas detecciones cuyo porcentaje de confianza sea superior al 40 %.

```
for i, score in enumerate(scores):
    if score > 0.4:
        label = labels[int(classes[i])-1]
        box = boxes[i]

        img_box = draw_boxes(img_box, box, label)

return img_box
```

Una vez conseguido el objetivo de optimizar el proceso, se decide dar un paso más y llevar a cabo un reentrenamiento de la red neuronal para obtener un modelo propio que sea capaz de detectar señales de tráfico, concretamente señales de stop y ceda el paso

3.2.2.2. Detección de objetos con modelo SSD reentrenado.

Cuando se entrena o comprueba un sistema como el desarrollado en este proyecto, es generalmente preferible usar datos de bases de datos públicas, en lugar de recolectar datos de forma particular. No sólo se ahorra tiempo y esfuerzo, además es conveniente para comparar resultados con trabajos anteriores en este área de estudio.

Se opta por reentrenar la red neuronal con imágenes de señales de tráfico debido a la facilidad de encontrar conjuntos de datos con miles de imágenes en internet, como sucede con el caso del conjunto de datos de señales de Tráfico Lisa (*Møgelmose y col., 2012).*

El trabajo trata de reentrenar el modelo con dos clases:

- Señal de stop. Tenemos 1821 para el reentrenamiento, como podemos ver en el listado de la figura 25.

- Señal de ceda el paso. Se dispone de un total de 293 imágenes para el reentrenamiento.

294	addedLane	34	slow
37	curveLeft	11	speedLimit15
50	curveRight	349	speedLimit25
35	dip	140	speedLimit30
23	doNotEnter	538	speedLimit35
9	doNotPass	73	speedLimit40
2	intersection	141	speedLimit45
331	keepRight	48	speedLimit50
210	laneEnds	2	speedLimit55
266	merge	74	speedLimit65
47	noLeftTurn	132	speedLimitUrdbl
26	noRightTurn	1821	stop
1085	pedestrianCrossing	168	stopAhead
11	rampSpeedAdvisory20	5	thruMergeLeft
5	rampSpeedAdvisory35	7	thruMergeRight
3	rampSpeedAdvisory40	19	thruTrafficMergeLeft
29	rampSpeedAdvisory45	60	truckSpeedLimit55
16	rampSpeedAdvisory50	32	turnLeft
3	rampSpeedAdvisoryUrdbl	92	turnRight
77	rightLaneMustTurn	236	yield
53	roundabout	57	yieldAhead
133	school	21	zoneAhead25
105	schoolSpeedLimit25	20	zoneAhead45
925	signalAhead		

In total: 7855 sign annotations

Figura 25. Contenido del conjunto de datos Lisa, numero de imágenes por clase. Møgelmose y col., 2012)

De acuerdo a *Stang (2017),* para reentrenar la red con TensorFlow hay que llevar a cabo una serie de pasos:

1. Elegir el modelo con el que vamos a trabajar. Como se citaba anteriormente, se va a trabajar con el modelo *ssd_mobilenet_v1_coco.*
2. Convertir el Dataset actual a TFRecord. Una vez elegido el modelo con el que se trabaja, necesitamos generar un archivo *tf.record*, que combinará tanto las imágenes como las etiquetas en las que se define dónde está el objeto buscado. El script recogido en el anexo IV se encarga de buscar aquellas imágenes del conjunto de datos con etiqueta stop y yield, de combinarla con sus anotaciones, las posiciones en las que están los objetos buscados, para finalmente generar un archivo con formato *.record*. En el caso particular de nuestro trabajo, generamos un archivo llamado stop_yield.record, ubicado en la carpeta *tf_record.* Usaremos este archivo, junto con otros necesarios para el reentrenamiento de la red neuronal.
3. Reentrenar el modelo con TensorFlow. Para ello seguimos los pasos recogidos en el segundo apartado del Anexo IV. Haremos uso del servicio en línea AWS por dos razones fundamentales, la primera por la velocidad de proceso que ofrece y la segunda para no poner en riesgo el trabajo de instalación que tenemos realizado en la GPU.
4. Convertir los modelos Checkpoints (.ckpt) a un archivo .pb

Estos pasos se detallan en el Anexo IV de la presente memoria. Asimismo, en el capítulo 5 podremos ver algunos detalles del resultado del proceso de reentrenamiento.

El reentrenamiento para las dos clases indicadas anteriormente, señal de stop y ceda el paso finaliza sin ningún tipo de error, no obstante, al probar el modelo congelado obtenido, se observa que no detecta ninguna de las dos clases reentrenadas.

Se decide, por tanto, usar un modelo proveniente del reentrenamiento de una sola clase, la de señal de stop. De igual forma, el proceso completo indicado anteriormente finaliza sin ningún tipo de error y en esta ocasión sí que se detecta la

clase reentrenada al ejecutar la red neuronal, como se verá en el siguiente capítulo de este trabajo.

En el caso de este modelo re-entrenado hacemos las pruebas con un factor de 0,7 para la confianza en las detecciones, así, el sistema sólo nos presentará como positivos y por tanto marcando con cajas de detección aquellas detecciones cuyo porcentaje de confianza sea superior al 70 %, la implementación concreta es la que se muestra en la secuencia de instrucciones siguientes:

```
# Cálculo de la inferencia. Presentamos las detecciones con puntuaciones superiores a 0,7

def inference(img):
    img_expanded = np.expand_dims(img, axis=0)


    (boxes, classes, scores) = sess.run([boxes_tensor, classes_tensor, scores_tensor], feed_dict={
      image_tensor: img_expanded
    })

    boxes, classes, scores = boxes[0], classes[0], scores[0]
    img_box = np.copy(img)
    for i, score in enumerate(scores):
        if score > 0.7:
            label = labels[int(classes[i])-1]
            box = boxes[i]

          t=time.time()

          print('Clase detectada:', label, 'Precision: {:.2%}'.format(score)) # Mostramos por
pantalla la clase detectada y la confianza

            img_box = draw_boxes(img_box, box, label) # Dibujamos las cajas y ponemos etiquetas

    return img_box
```

3.2.3. Planteamiento 3. Procesado con GPU y YOLO.

Una vez se ha conseguido detectar objetos con el planteamiento descrito en el apartado anterior, se desarrolla el sistema con otro tipo de red neuronal, concretamente se usará YOLO (You Only Look Once) *(Redmon y col, 2016)*, basado en Darknet *(Redmon, 2012)*, un entorno de redes neuronales de código abierto escrito en C y Cuda.

Se utilizará exactamente el mismo esquema de red que recoge la figura 23,. Al igual que sucedía en los planteamientos anteriores, el sistema estará integrado en ROS.

Para llevar a cabo la integración YOLO en ROS hacemos uso del repositorio *Gigioli (2017-2018)* . La diferencia fundamental con respecto al método utilizado en el

planteamiento anterior es que, en el caso de YOLO no utilizaremos un script, sino que instalaremos el paquete indicado y lo ejecutaremos en entorno ROS.

Se adapta el repositorio a nuestro trabajo, llevando a cabo modificaciones principalmente enfocadas a:

- Eliminar errores de compilación debidos principalmente al uso de software superior al recogido en los requisitos del repositorio -uso del JetPack 3.3 en la Jetson.
- Subscripción al tópico */frame* procedente del PC Virtual 1. Además, en origen, se tuvieron que asociar los parámetros altura y anchura al tópico para evitar errores (en *autopylot_agent.py* del PC Virtual 1).

Al igual que sucedía en el caso de TensorFlow, los primeros tests de YOLO se llevan a cabo con la imagen sin comprimir, y de igual forma se comprueba que el número de imágenes por segundo recibidas en la GPU en el tópico */frame* disminuye de forma relevante si lo comparamos con los fps que recibimos en PC Virtual 1. Por tanto, se sigue el mismo procedimiento de compresión de imágenes, consiguiendo, de igual forma, una notable mejora tanto en la recepción de imágenes como en la velocidad del proceso de detección.

La figura 26 muestra el esquema ROS para YOLO.

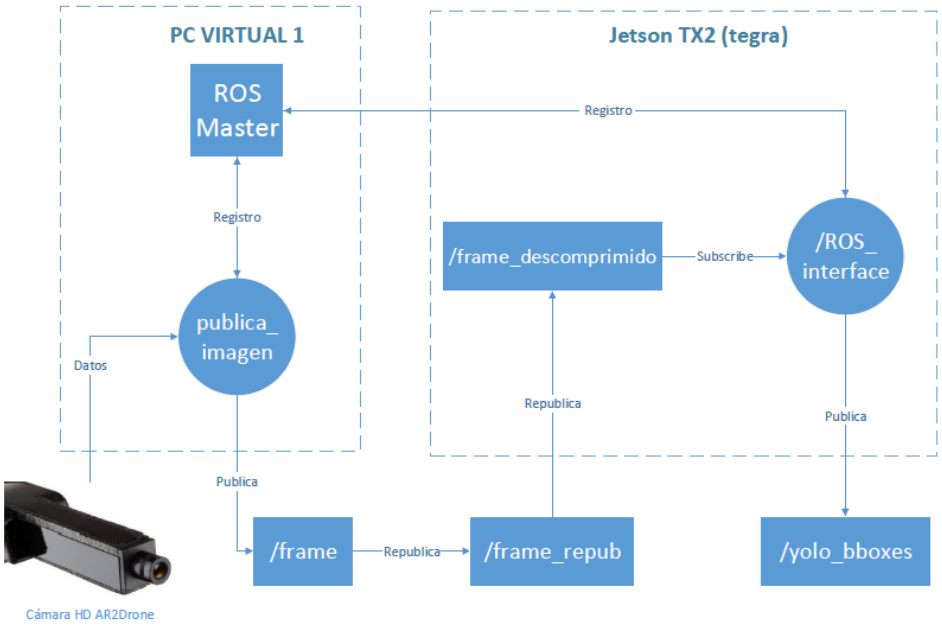


Figura 26. Esquema ROS para el sistema YOLO.

3.2.3.1. Detección de objetos basada en YOLO

La primera parte de las pruebas con YOLO integrado en el sistema se lleva a cabo con los pesos y el archivo de configuración predeterminados en el repositorio. Aún así hay que llevar a cabo unos pequeños ajustes, resumidos a continuación, sobre el archivo *yolo_ros.cpp*.

i. Definición de las variables de entrada para yolo.

```
// definimos las variables de entrada para yolo:
char *cfg = "/home/nvidia/catkin_ws/src/darknet_ros/yolo-voc.cfg";
char *weights = "/home/nvidia/catkin_ws/src/darknet_ros/weights/yolo-voc.weights";
float thresh = 0.5;
```

Vemos que el umbral de detección que utilizamos es de 0,5, esto es, aquellas detecciones que tengan un porcentaje de confianza superior al 50 % serán marcadas con cajas de detección y presentadas como positivas.

ii. Definición de las variables de entrada para YOLO.

```
// definir parámetros
const std::string CAMERA_TOPIC_NAME = "/frame_descomprimido";
const std::string CAMERA_WIDTH_PARAM = "/frame/image_width";
const std::string CAMERA_HEIGHT_PARAM = "/frame/image_height";
const std::string OPENCV_WINDOW = "YOLO object detection";
```

iii. Asegurar que el archivo yolo-voc.cfg está preparado para la detección.

Comprobamos que el archivo de configuración está configurado de la siguiente forma:

```
# Testing
batch=1
subdivisions=1
```

Una vez compilado el repositorio gracias al comando *catkin_make*, se procede a lanzar el sistema YOLO ROS mediante el comando: *rosrun darknet_ros yolo_ros*

3.2.3.2. Detección de objetos con modelo YOLO re-entrenado.

La segunda parte de las pruebas con YOLO se desarrolla con un modelo procedente de un re-entrenamiento llevado a cabo con imágenes de stop y ceda el paso.

Tras numerosas pruebas se decide trabajar directamente con el repositorio original *Darknet* (*Redmon, 2012)*. Se compila el repositorio para GPU y CUDA con el fin de obtener celeridad en el proceso.

Se sigue el proceso y se utilizan los conjuntos de datos y etiquetas de *Guanghan (2015).* Los principales pasos para realizar en reentreno son los siguientes: :

1. Preparar el dataset para el re-entrenamiento. Para ello se utiliza el script *process.py, Murugavel (2018).* Se modifica el código para obtener un 20 % de imágenes test y un 80 % de imágenes para el entrenamiento. Al ejecutar el script se generan dos archivos, *train.txt* y *test.txt*, que recogen un listado con la ubicación de cada imagen utilizada tanto para entrenamiento como para test. Es importante tener en cuenta que, cada imagen debe llevar asociada su etiqueta con el siguiente formato:

[número categoría] [centro objeto X] [centro objeto Y] [Ancho objeto en X] [Ancho objeto en Y]

2. Preparar el archivo *voc.data* para el reentrenamiento. Este archivo es base para el re-entrenamiento, dado que indica tanto las clases que pretendemos reentrenar como los archivos que vamos a utilizar para el reentrenamiento.

```
classes= 2
train  = /home/nvidia/Reentrenamiento/darknet/train.txt
valid  = /home/nvidia/Reentrenamiento/darknet/test.txt
names = data/voc.names
backup = /home/nvidia/Reentrenamiento/darknet/backup/
```

3. Crear el archivo *voc.names*, en el que definiremos las clases que queremos reentrenar, en nuestro caso el archivo tiene dos clases:

```
stopsign
yieldsign
```

4. Adaptación del archivo de configuración. Es de vital importancia modificar el archivo de configuración *yolo-voc.cfg* para prepararlo para el re-entrenamiento. Se llevan a cabo dos intentos iniciales; el primero de ellos con valores de 64/4 para el *batch* y *subdivisions* respectivamente y un segundo a 64/8. En ambos casos el reentrenamiento se ve interrumpido por una saturación de memoria RAM, así pues se decide usar los siguientes valores:

```
# Training
batch=64
subdivisions=16
```

Además es de vital importancia modificar el número de clases que queremos reentrenar, en este caso dos.

```
[region]
anchors =  1.3221, 1.73145, 3.19275, 4.00944, 5.05587, 8.09892,
9.47112, 4.84053, 11.2364, 10.0071
bias_match=1
classes=2
coords=4
num=5
softmax=1
jitter=.3
rescore=1
```

Al cambiar el número de clases a 2, tenemos que adaptar el filtro de la última capa convolucional con la siguiente fórmula,

$$Filters = num*(5+Número\ de\ Clases)$$

El valor que obtenemos es: 5*(2+5) = 35

```
[convolutional]
size=1
stride=1
pad=1
filters=35
```

```
activation=linear
```

Dejamos el resto de parámetros por defecto.

5. Ejecutar el comando de reentrenamiento. Se hace uso de los pesos pre-entrenados para las capas convolucionales obtenidos de *PESOS YOLO (2018).*

   El comando que se utiliza para reentrenar es:

./darknet detector train cfg/voc.data cfg/yolov2-voc.cfg darknet19_448.conv.23

La ejecución en la GPU nos permite monitorizar en todo momento el avance del proceso. En la figura 27 se puede observar el avance del re-entrenamiento. Podemos ver cómo el gráfico de la función de coste muestra un decrecimiento exponencial; asimismo vemos que la iteración número 1945 tiene un valor de coste de 0.06 y tenemos un promedio de valor de coste de reentrenamiento de 0.063, tras haber procesado 124480 imágenes.

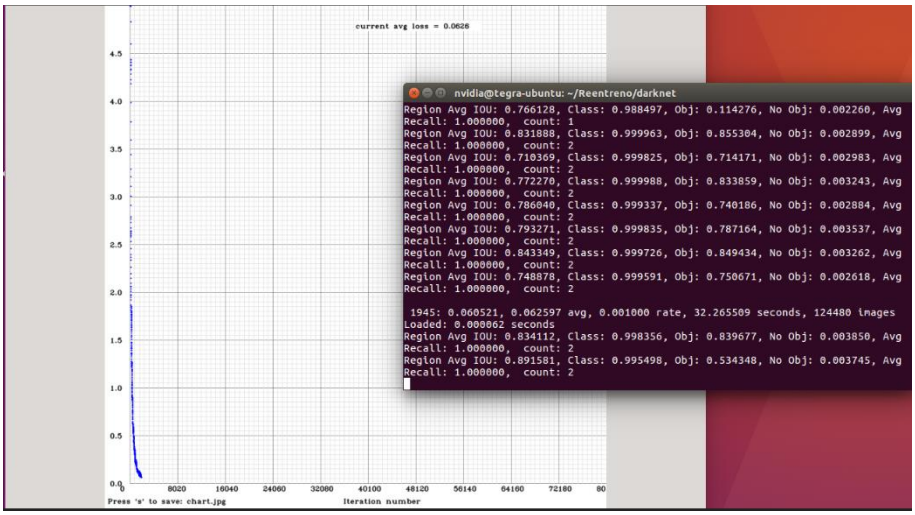


Figura 27. Reentrenamiento de Yolo con datos propios.

Tras finalizar el proceso de re-entrenamiento obtenemos un archivo con los pesos obtenidos.

Previamente a la ejecución del detector, se hacen unos últimos ajustes:

i. En el archivo de configuración *yolo-voc.cfg*, modificamos

```
# Testing
batch=1
subdivisions=1
```

ii. En el archivo *yolo_ros.cpp*, modificamos la ruta al archivo de configuración y a los pesos del sistema. Dejamos el umbral de detección en 0,5. Así, el porcentaje de confianza en la detección tendrá que ser superior al 50 % para que el sistema nos presente una detección de objeto como positiva.

```
// Usamos este para Yolo reentrenado
char *cfg = "/home/nvidia/catkin_ws/src/darknet_ros/cfg/yolo-voc.cfg";
char *weights = "/home/nvidia/catkin_ws/src/darknet_ros/weights/yolo-
voc_6100.weights";
float thresh = 0.5;
```

iii. En el archivo *yolo_ros.cpp*, modificamos las clases a detectar

```
//definimos las clases a usar
const std::string class_labels[] = { " stopsign", "yieldsign" };
```

iv. Volvemos a compilar el sistema YOLO-ROS gracias al comando *catkin_make.*

Finalmente el sistema se encuentra totalmente integrado y preparado para detectar con el modelo reentrenado. Los resultados obtenidos se estudian en el siguiente capítulo.

4. Capítulo 4. Resultados

Este capítulo recoge los resultados obtenidos en las pruebas planteadas, enfocadas a comparar los dos modelos objeto de estudio (SSD Mobilenet y YOLO) en el entorno de integración planteado en el apartado 3.2 de este trabajo.

Las pruebas se realizan en una pequeña parcela de unos 120 m^2, en la que se ubican algunos objetos como sillas, una bicicleta, botellas, monitores de ordenador y señales de tráfico de stop y ceda el paso. En el área además hay algunos árboles, plantas, personas y un perro. Los objetos que decidimos situar en la escena son objetos que ambos sistemas son capaces de detectar

En los primeros test de campo se observa, por un lado que controlar el dron con el driver planteado en *Levy, (2014, 2018)* y el joystick se hace realmente complicado en el espacio de que disponemos y por otro lado, en el caso de optar por llevar a cabo un vuelo por cada prueba, las comprobaciones no serían tan exhaustivas, ya que no se trataría del mismo recorrido y por tanto de los mismos datos.

En base a lo mencionado en el párrafo anterior, se opta por efectuar un vuelo con el dron gracias a la aplicación oficial de Parrot AR.FreeFlight 2 mientras se graba un video del recorrido utilizando la cámara frontal (HD 720p). El video grabado se procesa en el PC virtual 1 gracias a un script Python que lo reproduce y que además crea un nodo llamado *publica_imagen* que a su vez publica el tópico /frame. Podemos decir, de forma sencilla, que en lugar de procesar las imágenes del dron, procesamos el video grabado en el vuelo realizado. El resto de metodología es idéntica a la que se recoge en el apartado 3.2.

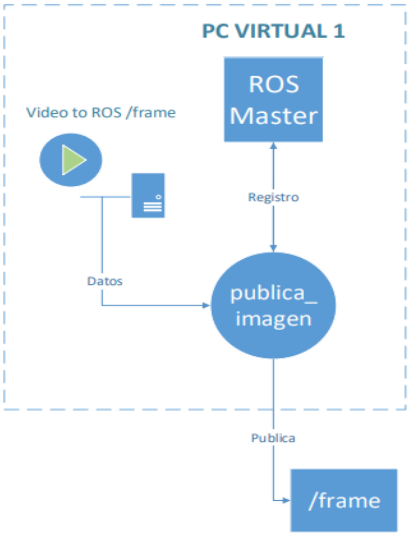


Figura 28. Estructura ROS del PC Virtual 1 para procesado en tests.

Al avanzar en la ejecución de las distintas pruebas se observa que al trabajar con el tópico */frame* procedente del procesado del video grabado obtenemos menos fps en la detección que si lo comparamos con los obtenidos en el tópico */frame* procedente directamente de la cámara del dron. Por esta razón se decide añadir dos pruebas más trabajando a tiempo real y capturando el resultado con la capturadora de video indicada en el Anexo I.

Estas son las pruebas llevadas a cabo durante el desarrollo de la parte práctica:

1. Prueba 1. Máquinas virtuales y SSD Mobilenet.
2. Prueba 2. GPU y SSD Mobilenet con /frame no comprimido
3. Prueba 3. GPU y SSD Mobilenet con /frame comprimido
4. Prueba 4. GPU y Yolo con /frame no comprimido
5. Prueba 5. GPU y Yolo con /frame comprimido
6. Prueba 6. GPU y Yolo reentrenado
7. Prueba 7. GPU y SSD Mobilenet reentrenado.
8. Prueba 8. GPU y Yolo reentrenado a tiempo real.
9. Prueba 9. GPU y SSD Mobilenet reentrenado a tiempo real.

En cada uno de los tests planteados se realizan pruebas de velocidad y de detección. En lo que respecta a la detección, la confianza se va a expresar en porcentaje. Para poder cuantificar ambas variables se guardan una serie de archivos de registro que incluyen:

- Registros de detección, que contienen la clase y porcentaje de confianza.
- Registros de fps en el tópico de detección.
- Registros del rendimiento de hardware (GPU + CPUs) (salvo en la prueba 1).

Además se captura la salida HDMI de la Jetson gracias a la capturadora de video indicada en el Anexo I, que se utilizará para la edición y montaje de una serie de videos preparados para analizar cualitativamente la detección.

A partir de los datos obtenidos de los archivos de registro se lleva a cabo un análisis cuantitativo de la detección. En el caso de los gráficos de fps y rendimiento de hardware obtenidos y dada la diferencia en la longitud en los archivos de registro, se lleva a cabo un suavizado de los datos, gracias al método estadístico *media móvil*

y referenciando en todo caso a 80 unidades, dado que es el número que encontramos que mejor encaja. Los valores en eje *x* no se mostrarán en los gráficos, dado que, debido al método utilizado, la precisión no es elevada y puede haber desfases en la secuencia de tiempo (no hemos registrado la hora exacta en los archivos de registro). El objetivo es ver la evolución a lo largo del tiempo más que el análisis puntual.

A continuación analizamos en profundidad los resultados obtenidos en cada una de las pruebas planteadas y la comparación entre ellas.

4.1. Prueba 1. Máquinas virtuales y SSD Mobilenet.

La tabla 2 muestra las clases detectadas en la prueba número 1, el número de detecciones que tienen lugar a lo largo del proceso para clase, así como el porcentaje de confianza promedio de las detecciones, también para cada clase. En la parte inferior se muestra el sumatorio del total de las detecciones y el promedio del porcentaje de confianza sobre el total de las detecciones.

Clase detectada	Nº Detecciones	% Confianza
Bicicleta	2	42,76
Bote	1	40,13
Botella	1	48,90
Brócoli	1	42,02
Tarta	20	48,85
Hidrante incendios	6	56,62
Horno	1	58,30
Persona	92	59,42
Maceta con planta	140	49,51
Señal de stop	39	65,97
Semáforo	1	41,92
Tren	8	47,68
Paraguas	1	42,58
Detección	**313**	**54,41**

Tabla 2. Clases detectadas en la Prueba 1.

Como puede observarse en la tabla 2, se obtienen 13 clases en un total de 313 detecciones. El promedio global de detección sobre las 313 detecciones es de 54,41%.

En el análisis de las imágenes procesadas, dos de cuyos ejemplos se muestran en las figuras 29 y 30, observamos que hay falsos positivos con clases que realmente aparecen en la escena, como sucede en la imagen 30, en la que se detecta una carretilla como si fuera clase *persona*. Vemos, asimismo, que existen otros objetos que ni siquiera aparecen en escena y que son detectados como clases en las imágenes de nuestro video. Así, las clases bote, brócoli, tarta, hidrante de incendios, horno, semáforo, tren y paraguas, son detectados en la escena, donde no aparecen, siendo también falsos positivos. Sólo teniendo en cuenta estas últimas clases citadas, obtenemos alrededor de un 12,5 % de falsos positivos.

Cabe destacar que, de forma general, los falsos positivos tienen menor porcentaje de confianza (ver tabla 2).

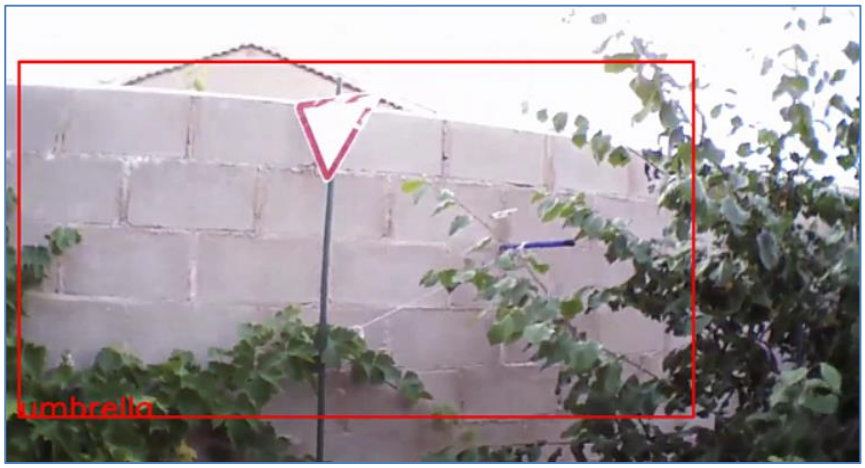


Figura 29. Falsa detección en Prueba 1.

Figura 30. Detección Prueba 1.

En lo que respecta al número de fps obtenidos en la imagen ya procesada observamos que es realmente bajo, empeorando aún más al capturar el video en entorno Windows (mayor uso de la RAM). La figura 31 muestra el gráfico de fps obtenidos de los datos del archivo de registro sin que se esté llevando a cabo captura de video. Con el fin de obtener una curva más suave, se efectúa un filtrado de los datos, como se menciona al principio del capítulo. Como puede observarse en el gráfico, el valor se estabiliza alrededor de 2,7 fps.

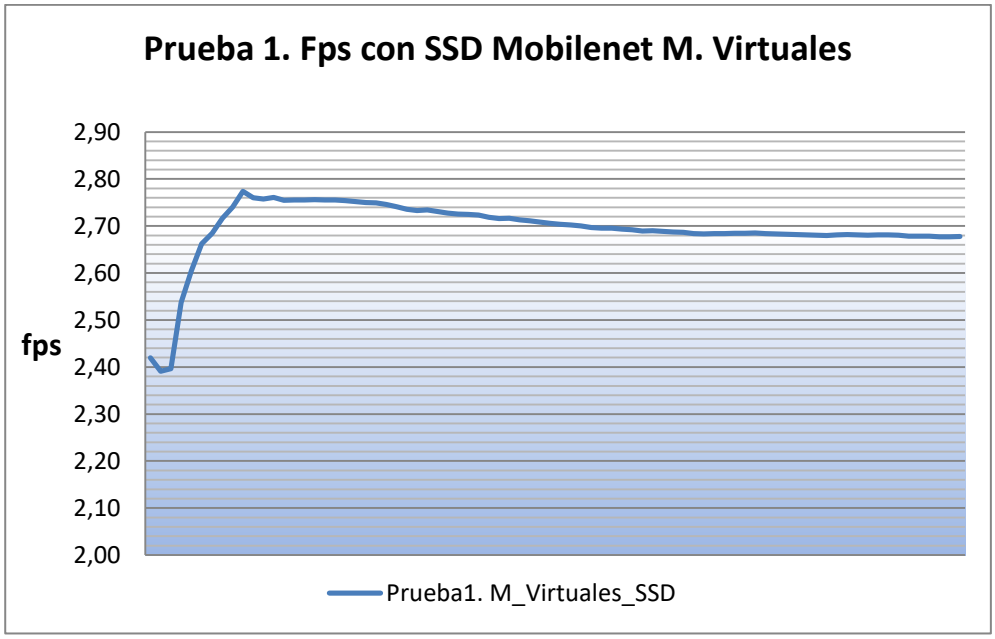


Figura 31. Fps en la imagen ya procesada en la prueba 1.

Como se mencionó previamente, en esta prueba no se registró el rendimiento del hardware.

Cualitativamente, observamos una gran lentitud y retraso en la imagen ya procesada. Vemos cómo gran parte de la escena original no aparece en el video ya procesado, quedando éste congelado durante grandes períodos de tiempo. También observamos que algunas clases no son detectadas en absoluto, como puede ser el ejemplo de la clase *perro* o la clase *tv* o bien no son detectadas con la persistencia que deberían, como sucede con la clase *botella.*

4.2. Prueba 2. GPU y SSD Mobilenet con /frame no comprimido.

En la segunda prueba que se realiza nos suscribimos al tópico /frame para procesarlo ya con la GPU. Como se comenta en el apartado 3.2, pese a que mejora ligeramente el número de fps en la imagen procesada, ni siquiera nos aproximamos al resultado esperado, como veremos en las próximas figuras. En la tabla 3 se muestran los resultados de la detección del sistema planteado en la prueba 2.

Clase detectada	Nº Detecciones	% Confianza
Bicicleta	82	86,18
Bote	22	60,72
Botella	2	45,75
Tarta	12	51,55
Silla	26	60,14
Copa	1	48,79
Hidrante incendios	6	51,75
Kite	12	47,79
Motocicleta	25	56,52
Persona	175	68,03
Maceta con planta	151	68,04
Señal de stop	175	72,53
osito de peluche	2	50,23
Semáforo	1	49,52
Tren	3	52,68
Paraguas	2	42,56
Detección	**697**	**69,28**

Tabla 3. Clases detectadas en la Prueba 2.

Como puede observarse en la tabla 3, se obtienen 16 clases de un total de 697 detecciones. El promedio global de detección es de 69,28 %. Los resultados, pese a no llegar a lo esperado, mejoran significativamente con respecto a los de la prueba 1, presentado más del doble de detecciones y prácticamente un 15 % más de confianza en la detección.

En el análisis de la imagen procesada, vemos que se detecta una clase más que no se detectaba en la prueba anterior (verdadero positivo), la clase silla. Observamos que, de igual forma, hay falsos positivos de clases que realmente aparecen en la escena,

aunque no se analizan en profundidad porque resulta un proceso tedioso para la obtención de un valor cuantitativo poco valioso.

Vemos que también existe detección de objetos que no están en la escena y son detectadas como clase, así, en la prueba 2 no detectamos las clases brócoli y horno que sí se detectaban en la prueba 1. Se detectan las clases bote, tarta, hidrante de incendios, semáforo, tren y paraguas en común en ambas pruebas. El segundo test detecta, además, las clases copa, kite, osito de peluche y motocicleta (que se confunde con la bicicleta en la detección). El porcentaje de detección de estos objetos que no están en la escena (falsos positivos) ronda el 12,35 %, prácticamente el mismo valor que presentaba la prueba anterior. Cabe aclarar que, si dejamos fuera del cálculo la clase motocicleta (se ve en el video claramente que el sistema confunde la bicicleta con una motocicleta por su morfología), ese porcentaje baja al 8,75 %.

Respecto al número de fps que obtenemos en la imagen procesada, se observa a partir de la gráfica mostrada en la figura 32 que sube significativamente con respecto a la prueba 1, quedándose en torno a 4,25 fps, como puede apreciarse en la figura 32.

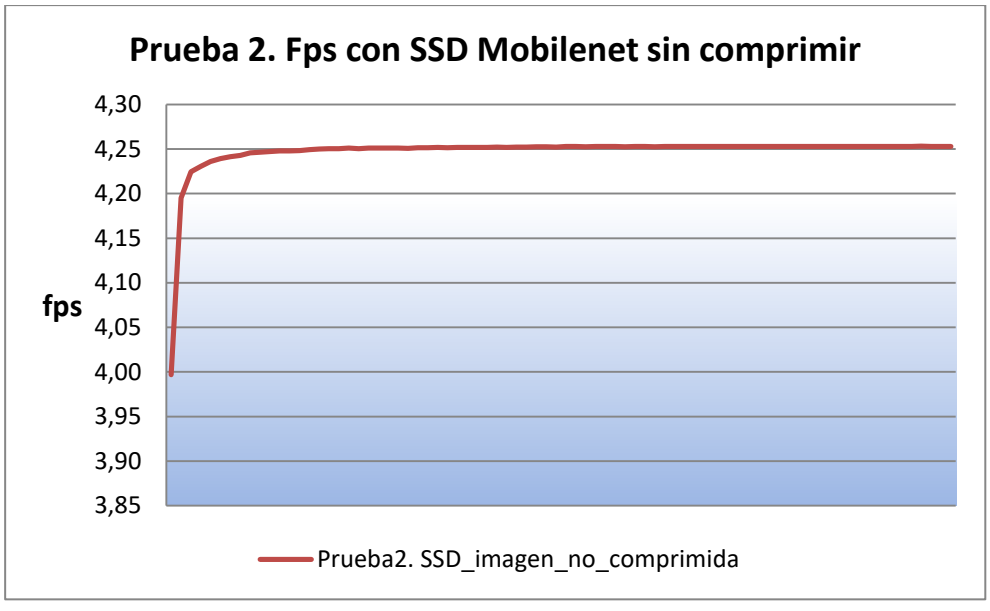


Figura 32. Fps en la imagen ya procesada en la prueba 2.

En lo que respecta al rendimiento del hardware, se observa a partir de los gráficos correspondientes, figuras 33 y 34, tanto al comportamiento de la GPU como a las

seis CPUs que presenta la Jetson TX2. Vemos que el rendimiento de la GPU es en general bajo.

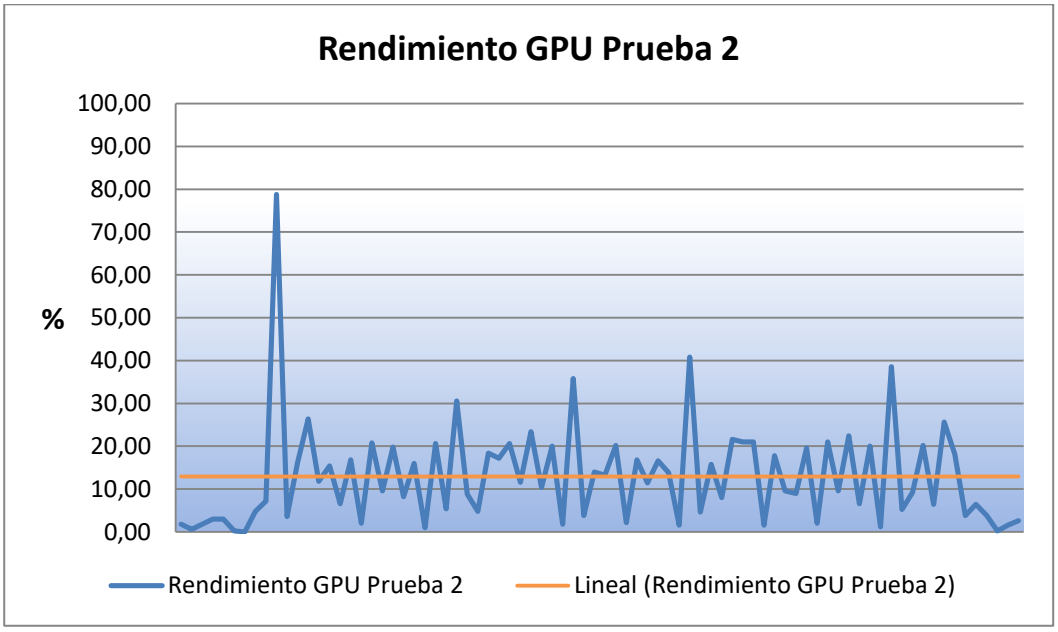


Figura 33. Rendimiento GPU en la Prueba 2.

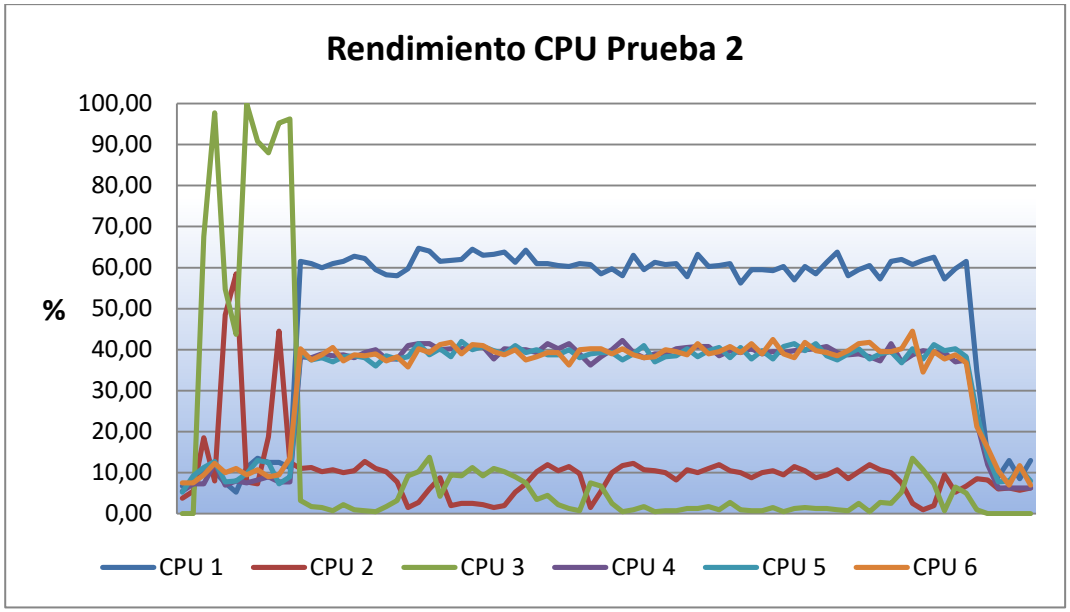


Figura 34. Rendimiento CPUs en la Prueba 2.

Desde un punto de vista cualitativo, observamos una leve mejora en lo que respecta al retraso de la imagen procesada con respecto al video original.

Pese a que hay partes del video que ni siquiera se llegan a visualizar en el video que muestra la detección, como sucedía en la prueba anterior, el video obtenido del

procesamiento de la detección es moderadamente más fluido (entendiendo por fluido una visualización de video que avanza con facilidad, sin interrupciones) y la imagen no se queda tan congelada como sucedía en la prueba anterior. Respecto de los objetos detectados, en la figura 35 vemos que las clases persona y planta son detectas de forma precisa y que además las cajas presentan unas coordenadas bastante exactas en la detección. Sin embargo, la clase botella no ha sido detectada por el sistema.

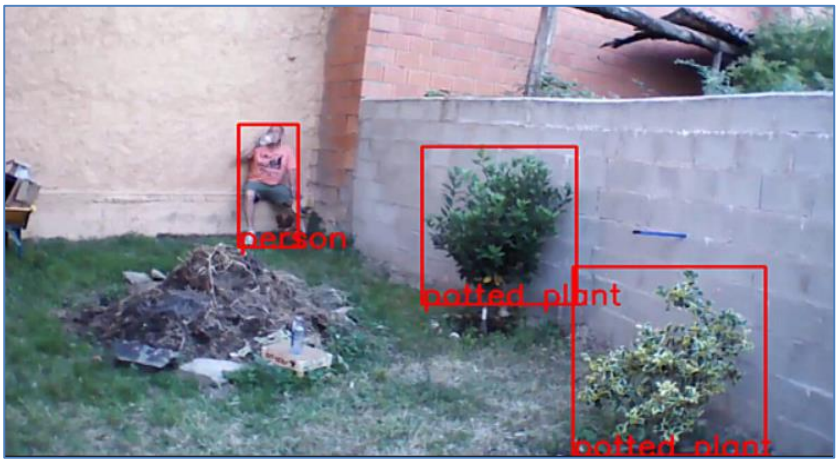


Figura 35. Detección de clases planta y persona en la prueba 2.

También observamos que algunas clases no son detectadas en absoluto, como puede ser el ejemplo de la clase *perro* o la clase *tv* o no son detectadas con la persistencia que deberían, como sucede con la clase *botella*, figura 36

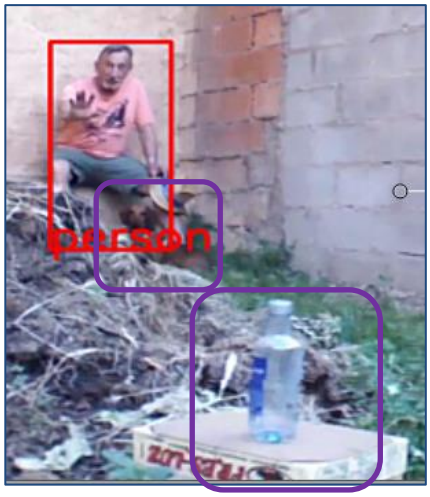


Figura 36. Clases no detectadas en la prueba 2. Clase perro y clase botella (encuadradas en violeta).

4.3. Prueba 3. GPU y SSD Mobilenet con /frame comprimido.

La prueba 3 se desarrolla comprimiendo las imágenes transportadas en el tópico */frame* desde el PC Virtual 1 hasta la Jetson TX2. La mejora es notable en lo que respecta a velocidad de proceso y por tanto en los fps obtenidos en las imágenes ya procesadas transportadas en el tópico */frame_box*.

La tabla 4 muestra los resultados de detección obtenidos en la prueba 3.

Clase detectada	Nº Detecciones	% Confianza
Bicicleta	286	77,62
Bote	14	55,38
Tarta	38	53,50
Silla	58	59,8%
Hidrante incendios	14	56,11
Frisbee	3	53,83
Bolso de mano	1	43,97
Caballo	3	58,28
Kite	10	47,53
Motocicleta	55	58,14
Horno	2	50,52
Persona	609	62,80
Planta	435	59,16
Señal de stop	351	76,00
osito de peluche	17	48,64
Semáforo	17	53,04
Tren	12	55,78
Camión	6	47,33
Tv	1	57,21
Paraguas	4	52,07
Florero	4	49,66
Detección	**1940**	**65,58**

Tabla 4. Clases detectadas en la Prueba 3.

En la prueba 3 se obtienen 21 clases de un total de 1940 detecciones. El resultado en el número de detecciones de esta prueba multiplica el obtenido en la prueba 2 por más de 2,7, así pues, la mejora es significativa.

El promedio global de detección es de 65,58 % frente al 69,28 % de la prueba 2. El porcentaje de falsos positivos (sin tener en cuenta los falsos positivos de clases que sí se encuentran en la escena), es de un 10,31 %, teniendo en cuenta la clase motocicleta y de un 7,47 % si la excluimos del cálculo.

En cuanto a las clases detectadas, comprobamos que la prueba 3 detecta 6 falsos positivos más que la prueba 2, entre las que se incluyen las clases *frisbee, bolso de mano, caballo, horno, paraguas* y *florero*. También detecta un verdadero positivo más, la clase *tv*. La prueba 2, por su parte detecta un verdadero positivo que no detecta la prueba 3, la clase *botella* y un falso positivo, la clase *copa*.

En lo que respecta al número de fps obtenidos en el tópico */frame_box*, comprobamos que los resultamos mejoran considerablemente desde el punto de partida y también con respecto a la prueba 2, presentando valores próximos a 8,6 fps, figura 37

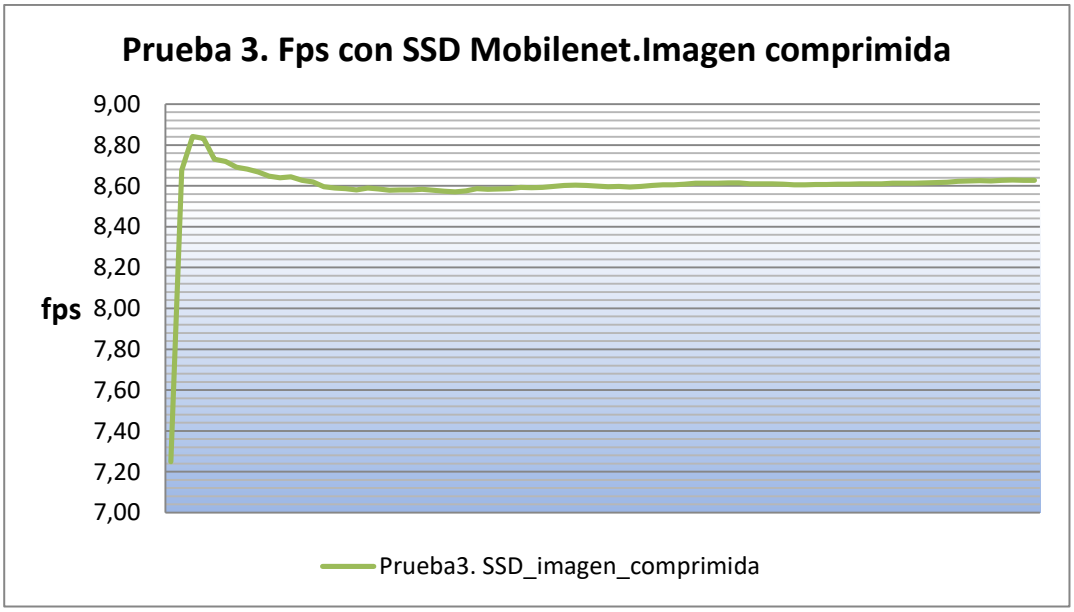


Figura 37. Fps en la imagen ya procesada en la prueba 3.

El porcentaje de funcionamiento de la GPU durante el proceso aumenta de forma moderada, como podemos comprobar en la línea de tendencia del gráfico mostrado en la figura 38 si la comparamos con el de la figura 33. También podemos ver un aumento en el porcentaje de funcionamiento de las CPUs. Esto probablemente se deba al aumento en la cantidad de datos a procesar por el hardware. Al usar la

compresión de imágenes se aumenta en el número de imágenes por segundo que alimentan la entrada de la red neuronal y por tanto el uso de GPU y CPUs, como puede comprobarse en las figuras 38 y 39.

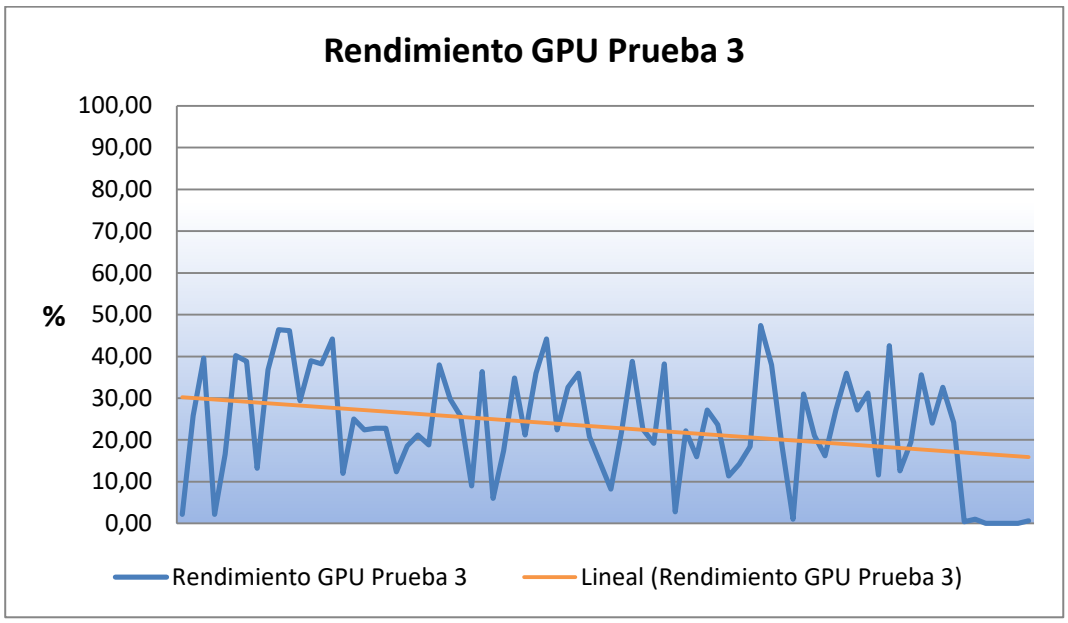


Figura 38. Rendimiento GPU en la Prueba 3

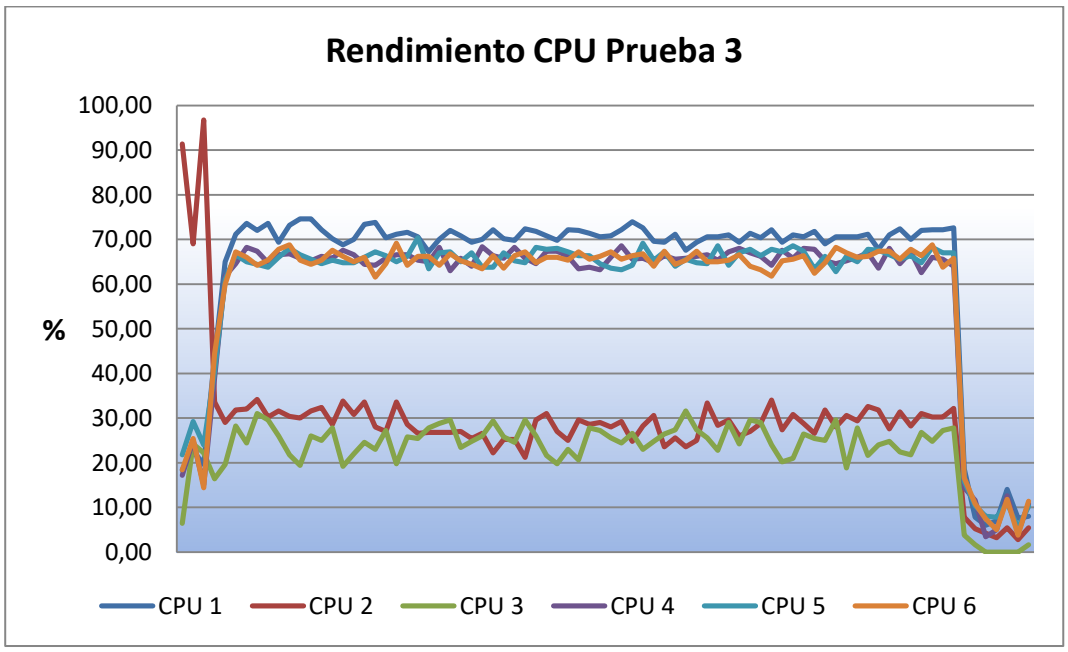


Figura 39. Rendimiento CPUs en la Prueba 3.

Desde un punto de vista cualitativo aún se observa un cierto retraso de la imagen procesada con respecto a la original, aunque, dentro de ese retraso, la mayor parte de la escena se visualiza de forma correcta.

Vemos además que se siguen presentando prácticamente los mismos errores de detección que se observaron en las pruebas anteriores. Lo mismo sucede con los objetos existentes en la escena y no detectados, así, las clases *perro, botella, tv*…o bien no son detectadas o bien se detectan en muy baja medida.

En lo que respecta a la distancia, observamos que los objetos que están a una distancia relativamente alejada, como sucede con los objetos bicicleta, botella y señal de stop, encuadrados en naranja en la figura 40, no son detectados por el sistema.

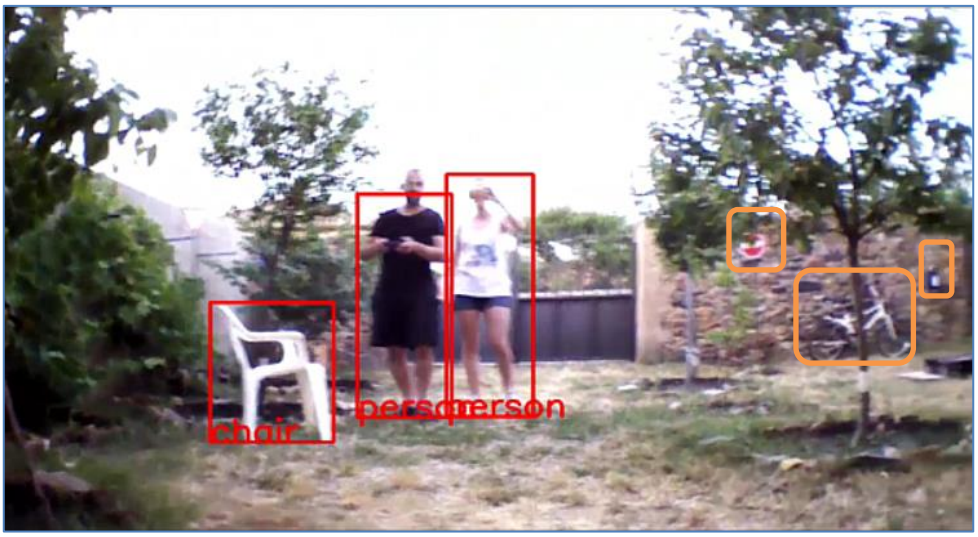


Figura 40. Detección de persona y silla en la prueba 3.

4.4. Prueba 4. GPU y Yolo con /frame no comprimido.

Como se veía en el capítulo 3, el modelo Yolo que se ha usado es capaz de detectar 20 clases mientras que ssd_mobilenet_v1_coco puede detectar hasta 90 clases. Como se comentaba al principio de este capítulo, los objetos que decidimos poner en la escena son objetos que ambos sistemas son capaces de detectar. La limitación en la detección de Yolo a 20 clases hace que el número de detecciones sea inferior, fundamentalmente debido a que no está detectando la señal de Stop.

Clase detectada	Nº Detecciones	% Confianza
Bicicleta	17	60,08
Silla	18	68,51
Persona	206	64,76
Planta	22	62,43
Oveja	1	51,99
Detección	**264**	**64,47**

Tabla 5. Clases detectadas en la Prueba 4.

En la tabla 5 vemos que obtenemos 5 clases distintas en 264 detecciones, obteniendo un promedio de 64,47 % en la confianza de detección. Tenemos únicamente un falso positivo en lo que respecta a objetos que no están en la escena, la clase oveja, que supone un porcentaje inferior al 1 % sobre el total de detecciones. Igual que sucedía en las pruebas anteriores con SSD Mobilenet, en Yolo también aparecen falsos positivos en clases que existen en la escena, aunque no entraremos en la cuantificación de los mismos. Como se puede comprobar en la figura 41, exactamente igual que sucedía con el modelo SSD Mobilenet, se detecta la carretilla como clase *persona*.

También puede observarse que determinados objetos situados a una cierta distancia no son detectados, como sucedía también en SSD Mobilenet. Puede confirmarse esta afirmación en la imagen de la figura 42, en la que vemos cómo la bicicleta -encuadrada en naranja- no es detectada, mientras que las clases que están más cercanas, *persona* y *silla*, son detectadas perfectamente.

Figura 41. Falso positivo en prueba 4.

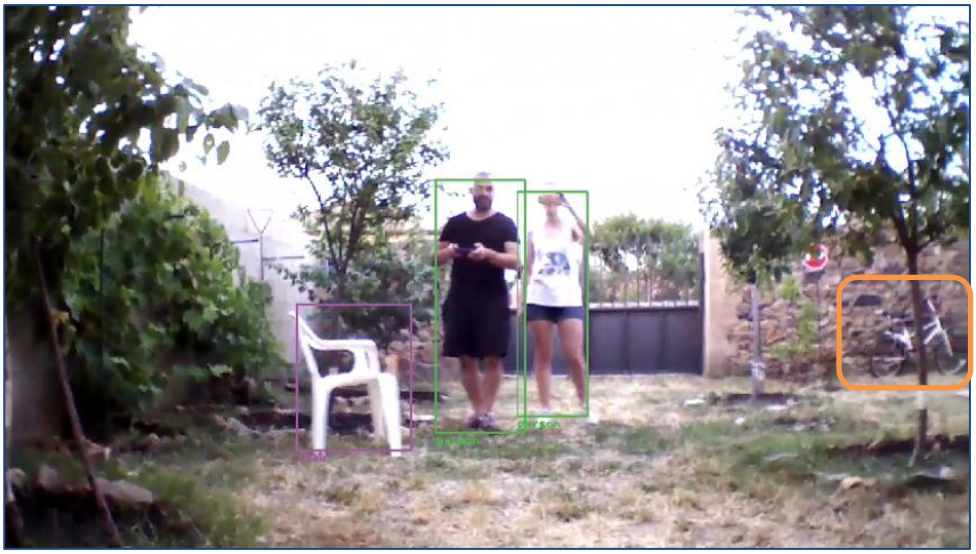


Figura 42. Detección de persona y silla en la prueba 4.

El número de fps que obtenemos en el tópico YOLO_bboxes, que transporta la imagen tras la detección con Yolo es en todo caso inferior a 2,5. Como se comenta en el apartado 3.2, este bajo ratio se debe fundamentalmente a que el número de fps que se reciben en el tópico */frame* en la Jetson TX2, que es al que nos suscribimos para llevar a cabo la detección es realmente bajo, debido al tamaño de las imágenes transportadas. La figura 43 muestra el número de fps de la imagen procesada a lo largo de la detección de Yolo con las imágenes transportadas en el tópico /frame sin comprimir.

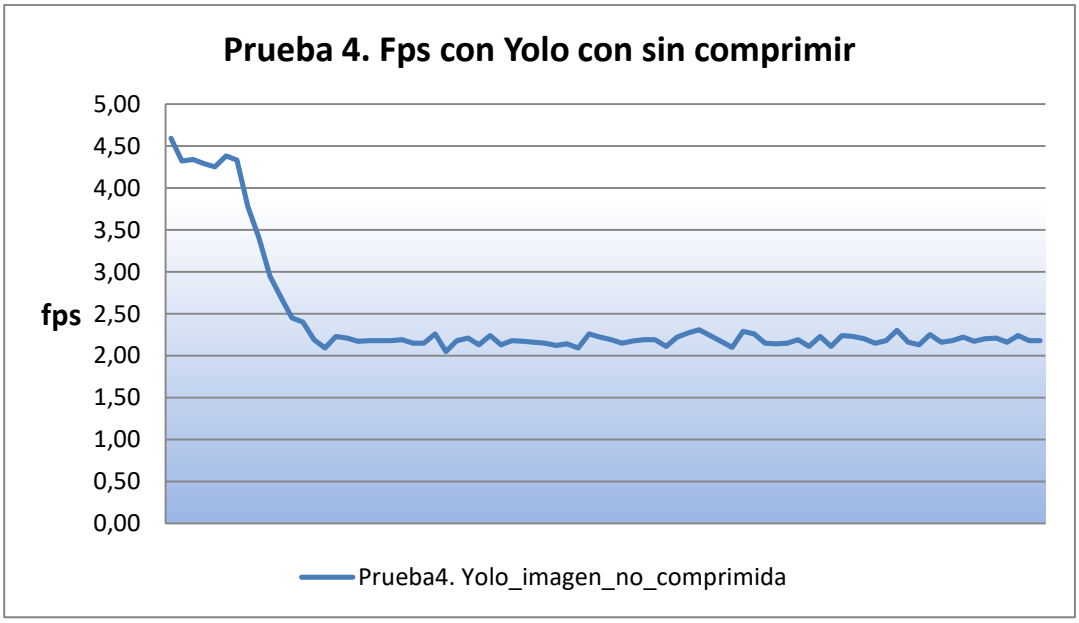


Figura 43. Fps obtenidos en la prueba 4.

Con respecto al rendimiento del hardware, tal y como podemos observar en las figuras 44 y 45, podemos decir que, de forma general, se observa un mayor porcentaje de funcionamiento de la GPU y menor en las CPUs. En el apartado 4.10, se proporcionan detalles adicionales a este respecto.

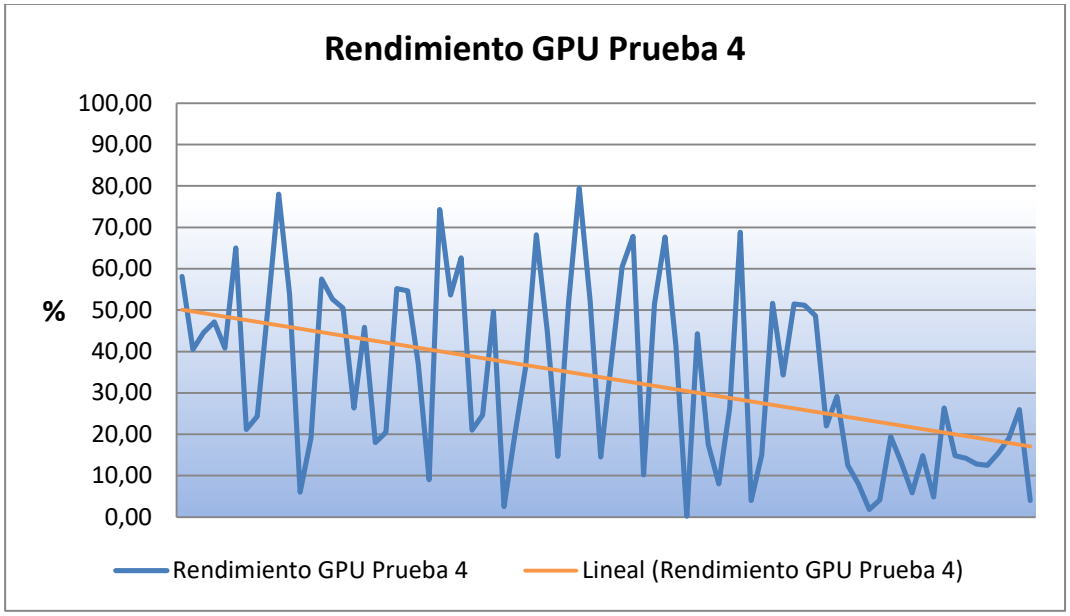


Figura 44. Rendimiento GPU Prueba 4.

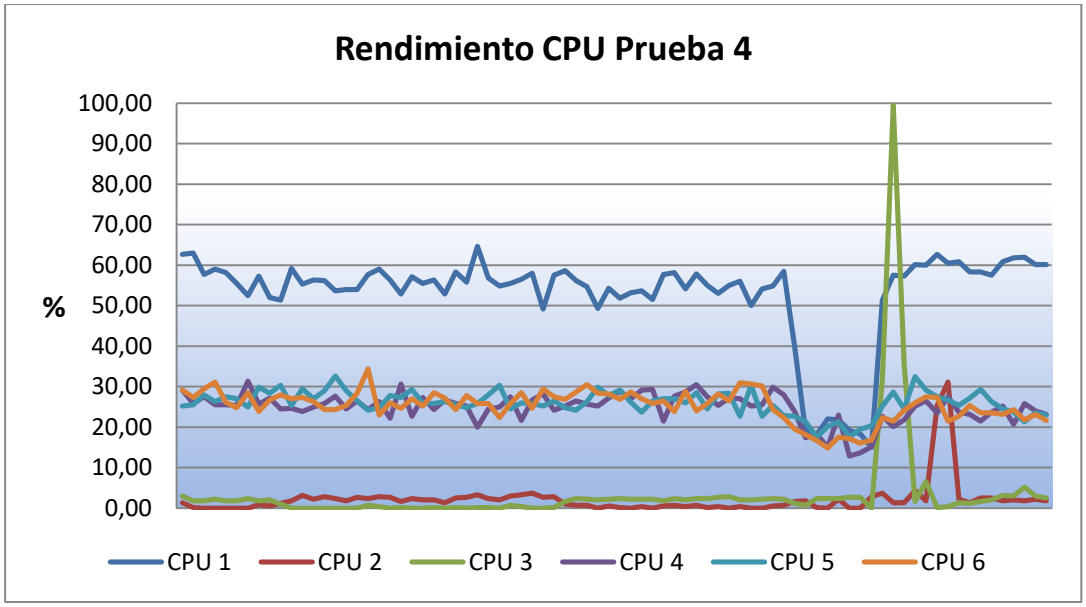


Figura 45. Rendimiento CPUs Prueba 4.

Desde un punto de vista cualitativo, observamos que, como sucedía en la prueba 2, las imágenes obtenidas no son fluidas, llegándose incluso a quedar la imagen congelada, por lo que no toda la escena se muestra si la comparamos con el video original.

4.5. Prueba 5. GPU y Yolo con /frame comprimido.

La prueba número cinco se lleva a cabo comprimiendo el tópico /frame en el PC Virtual 1 y descomprimiendo en otro tópico ya en el sistema embebido Jetson TX2. De esta forma y debido al número de fps recibidos en el tópico */frame_descomprimido* -30 fps, como en el PC Virtual 1- se consigue una notable mejora en lo que respecta a velocidad de proceso en la detección, siendo también así en el número de detecciones. Si comparamos el número de detecciones con respecto a la prueba 3, se obtiene un número considerablemente inferior, achacable al menor número de clases con que trabajamos en Yolo, pero también en el menor número de fotogramas que obtenemos en esta prueba, como se indica seguidamente. La tabla 6 muestra un resumen de la detección obtenida en esta prueba.

Clase detectada	Nº Detecciones	% Confianza
Bicicleta	168	74,16
Botella	7	63,97
Coche	1	53,95
Silla	75	75,57
Motocicleta	7	70,01
Persona	309	64,55
Planta	38	59,23
Oveja	2	74,42
Tren	1	67,65
Monitor tv	1	53,50
Detección	**609**	**68,28**

Tabla 6. Clases detectadas en la Prueba 5.

Se obtienen 10 clases a partir de 609 detecciones con un promedio en la confianza de detección del 68,28 %. Este valor mejora en torno a un 3,8 % y el número de detecciones es más del doble que en el modelo Yolo que usa imágenes sin comprimir. El número de clases detectadas en esta prueba es el doble que las detectadas en la prueba 4. Al comprimir las imágenes del tópico */frame* hemos conseguido detectar dos clases que existen en la escena y que no se detectaban en la prueba anterior, la clase *botella* y la clase *Monitor tv*. Del resto de clases nuevas detectadas, *coche* y *tren* no aparecen en la escena y en el caso de *motocicleta* y como sucedía con SSD, vemos que el sistema lo confunde con la bicicleta. El número de

falsos positivos de objetos que no existen en la escena se sitúa en torno al 1,8 % si incluimos la clase motocicleta en inferior al 1% si la excluimos del cálculo. De igual forma y como sucedía en la prueba 4, también existen falsos positivos de objetos que aparecen en la escena (carretilla detectada como persona).

La figura 46 muestra el número de fps obtenidos en el tópico YOLO_bboxes, que contiene la imagen ya procesada. Observamos que toma valores entre 4,55 y 4,90, sin sobrepasar en ningún momento este valor. El resultado mejora substancialmente con respecto al obtenido en la prueba 4, pero es considerablemente inferior (casi la mitad de fps) que el obtenido con SSD Mobilenet en combinación con la compresión de imágenes.

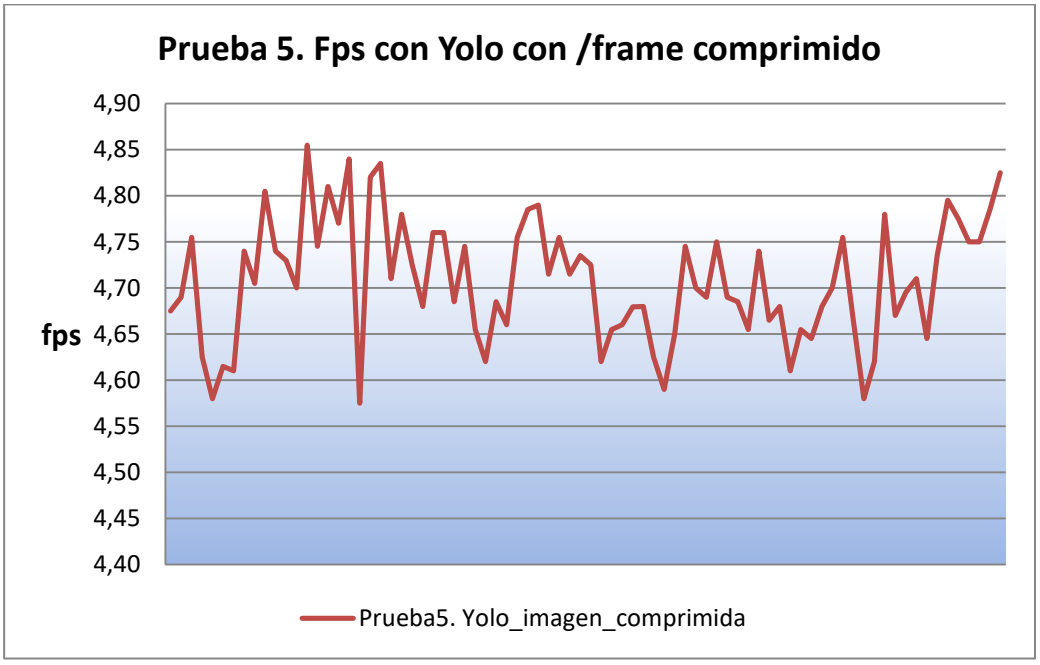


Figura 46. Fps obtenidos en la prueba 5.

Cualitativamente observamos que el video ya procesado es fluido y que la detección es buena, con las limitaciones anteriormente expuestas. Si lo comparamos con los resultados obtenidos en la prueba 3, vemos menor retraso entre el video original y el video procesado en la prueba 5.

En lo que respecta al rendimiento de hardware, observamos que el porcentaje de uso de la GPU aumenta notablemente con respecto a la prueba 4 y es en gran medida mayor que el que obtenemos en la prueba 3. Con respecto al uso de las CPUs, observamos que el porcentaje de uso obtenido en la prueba 5 es inferior tanto

si lo comparamos con la prueba 3 como con la prueba 4. Las figuras 47 y 48 muestran los gráficos del rendimiento de hardware para la prueba 5.

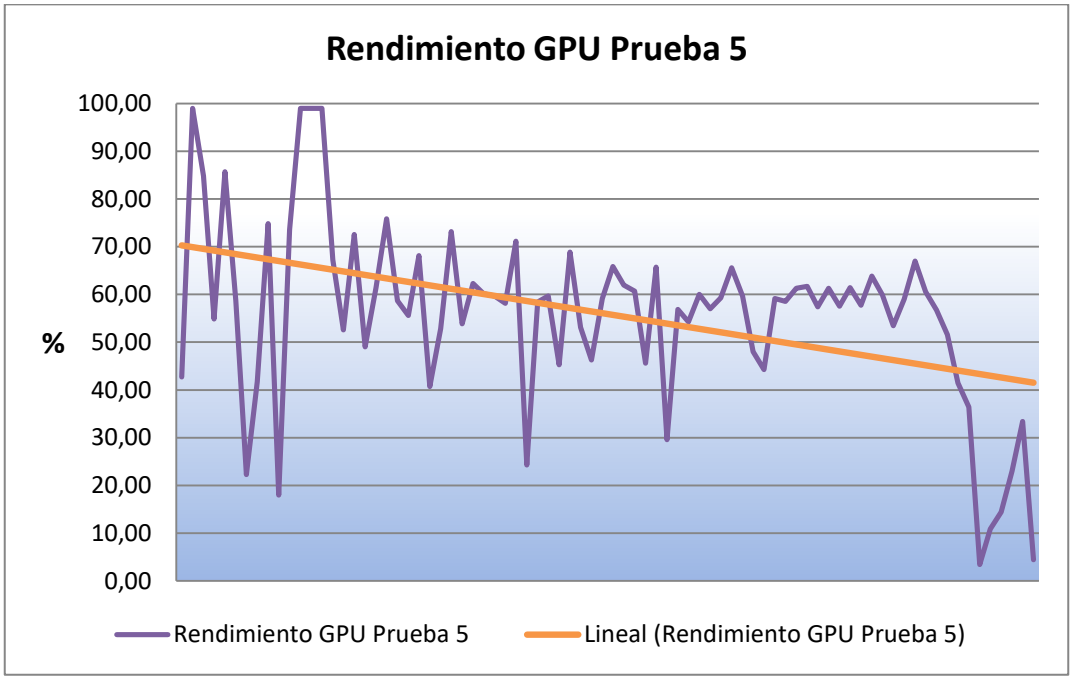


Figura 47. Rendimiento GPU Prueba 5.

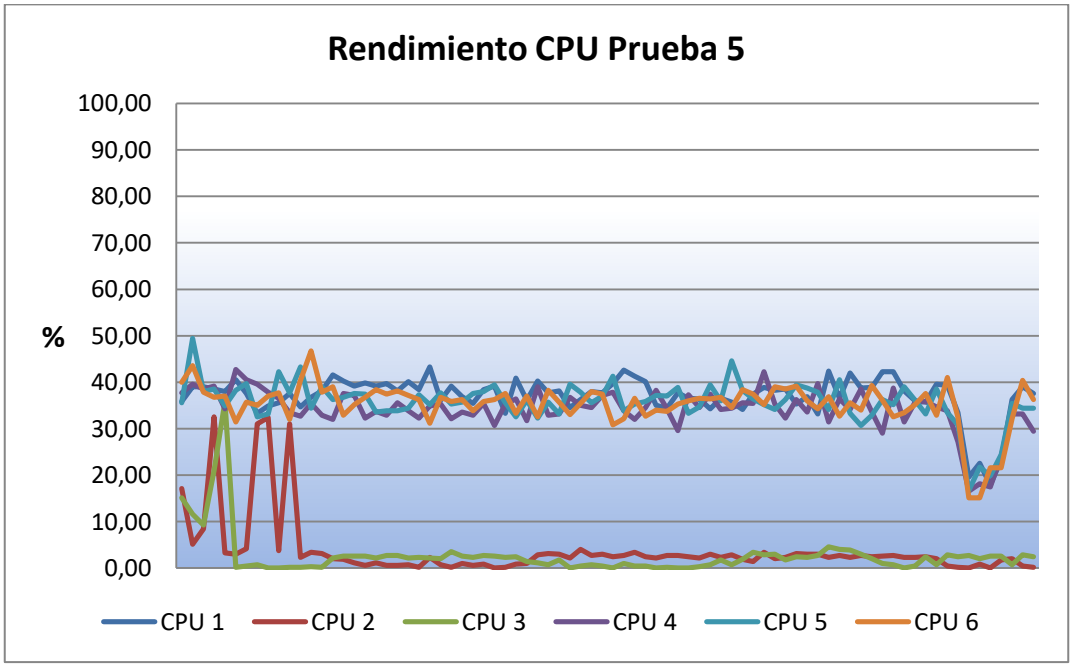


Figura 48. Rendimiento CPUs Prueba 5.

4.6. Prueba 6. GPU y Yolo reentrenado.

Una vez optimizado el sistema gracias a la compresión de imágenes procedentes del tópico */frame* se decide dar un paso más, realizando un re-entrenamiento de la red neuronal haciendo uso de un conjunto de datos de señales de tráfico ya etiquetados para usar en Yolo (*Guanghan, (2015))*. El re-entrenamiento se lleva a cabo con dos objetivos fundamentales, el primero es abrir la posibilidad de usar nuestro sistema con cualquier conjunto de imágenes que decidamos y el segundo es investigar si el proceso se ve optimizado y se obtienen aún mayor número de fps. Como se menciona en el apartado 3.2, el re-entrenamiento se lleva a cabo sobre dos clases: *señal de ceda el paso* y *señal de stop*. La tabla 7 recoge los resultados obtenidos tras procesado de la imagen.

Clase detectada	Nº Detecciones	% Confianza
Señal de ceda el paso	181	71,05
Señal de stop	194	72,39
Detección	**375**	**71,74**

Tabla 7. Clases detectadas en la Prueba 6.

Las dos clases reentrenadas se detectan en un total de 375 ocasiones en la escena, presentando un porcentaje de confianza promedio de 71,74 %, el cual supera el resultado de cualquiera de las pruebas anteriores en lo que respecta a confianza de detección.

En lo que respecta al número de fps obtenidos en el tópico YOLO_bbox, vemos que no hay una gran diferencia si lo comparamos a los resultados obtenidos para esta variable para la prueba anterior. El número oscila entre 4,6 y 4,85 fps, valores muy similares a los obtenidos en la prueba 5, como puede comprobarse en la figura 49.

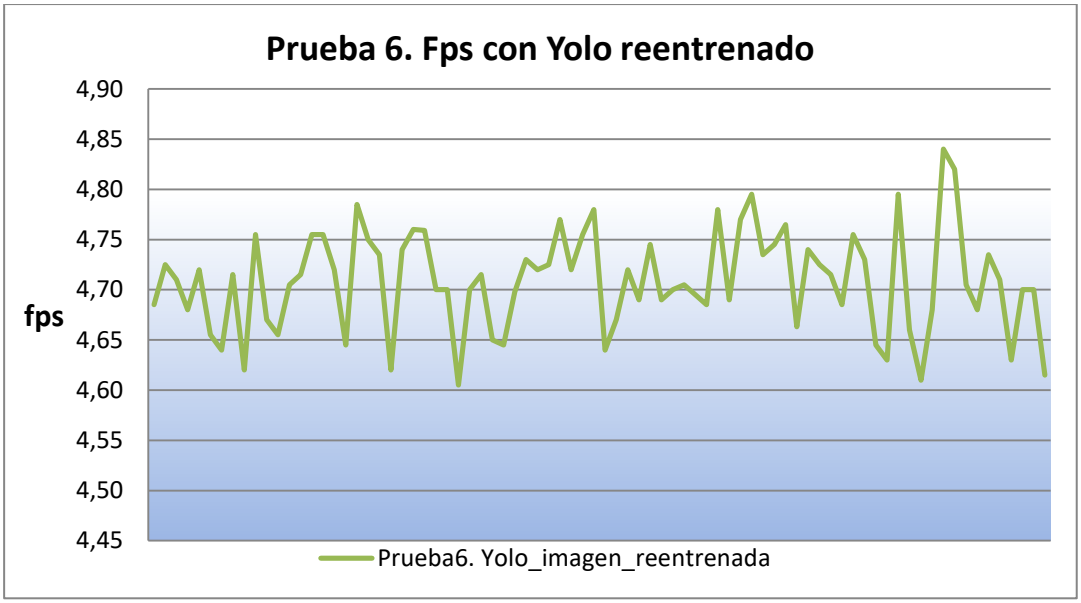


Figura 49. Fotogramas por segundo obtenidos en la prueba 6.

Con respecto al rendimiento de la GPU, pese a que el comienzo arroja datos relativamente inferiores con respecto a la prueba anterior, vemos cómo tiene una mejor persistencia a lo largo del proceso. Así pues, no se aprecian grandes diferencias en el promedio global, como podremos comprobar posteriormente en la comparativa global. La figura 50 muestra el gráfico obtenido en el rendimiento de la GPU en la prueba 6.

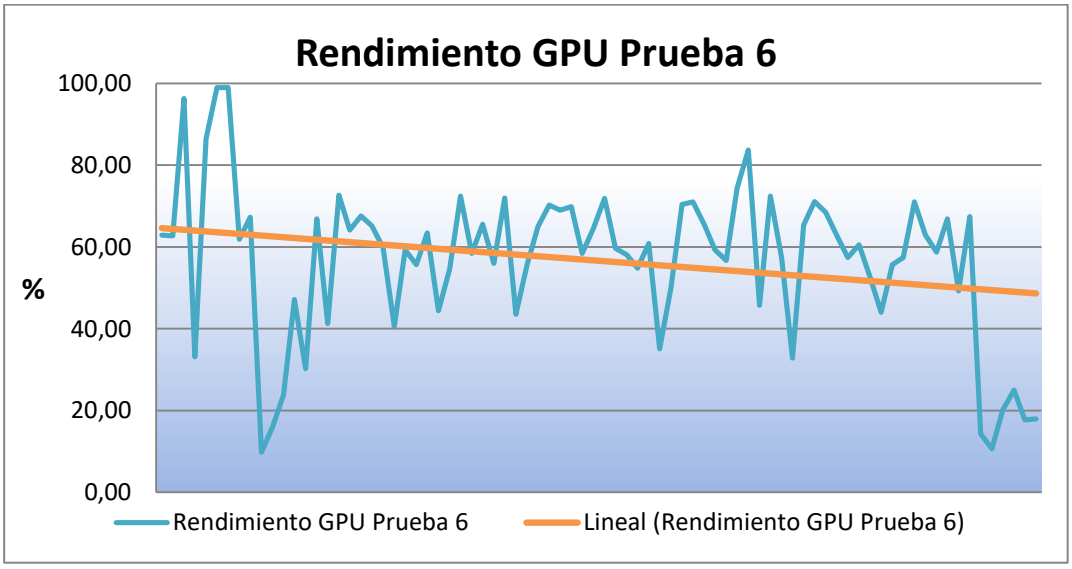


Figura 50. Rendimiento GPU Prueba 6.

En lo que respecta al porcentaje de funcionamiento de las CPUs, vemos que éstas están funcionando de una forma moderada, presentando resultados similares a los que obteníamos en la prueba anterior. Los porcentajes de las CPU 2 y 3 presentan

valores promedios inferiores al 10 % mientras que las CPUs 1, 4, 5 y 6 presentan valores que oscilan entre el 30 y el 50 % como promedio, como puede observarse en la figura 51.

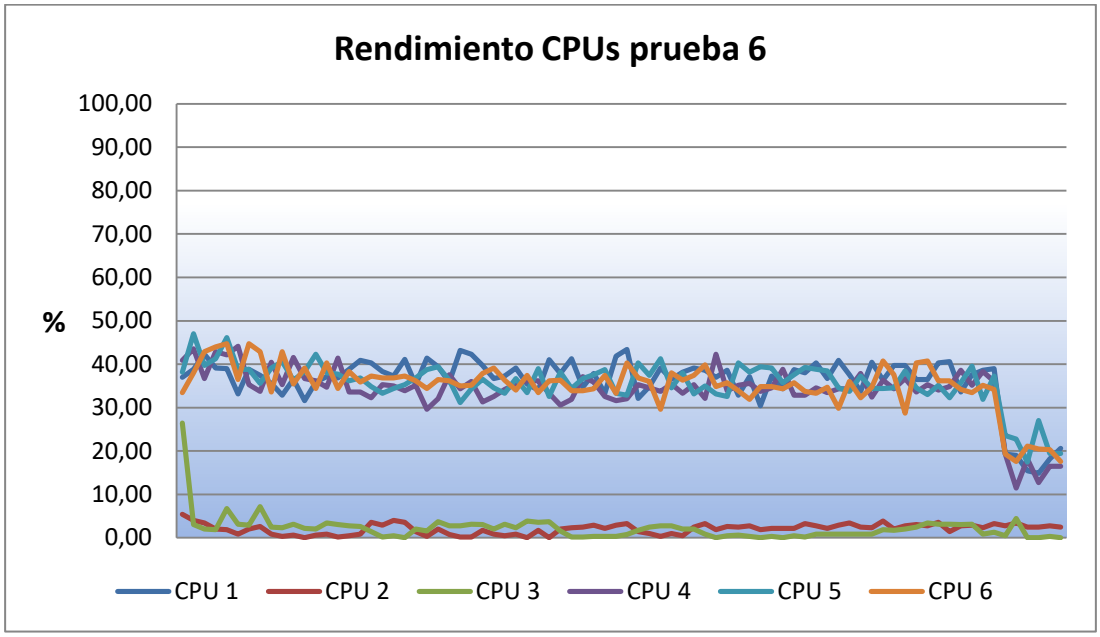


Figura 51. Rendimiento CPUs Prueba 6.

Cualitativamente, observamos fluidez en el avance de las imágenes, aunque en alguna ocasión se observan desfases con respecto al video original. En cuanto a la detección, no parece haber falsos positivos, sin embargo, el sistema no detecta algunas de las señales con la persistencia que se desearía. El dron detecta las imágenes mejor cuando se encuentra próximo a la señal y a la misma altura que ésta, posiblemente influya el brillo que presentan las señales desde ciertos ángulos, como puede comprobarse en la figura 52. También vemos que la señal de ceda el paso se reconoce mejor en la escena, como podemos comprobar en la imagen de la figura 53, en la que se ve cómo se detecta la señal de ceda el paso, pero no se detecta la señal de stop.

Figura 52. Detección de señal de stop con YOLO reentrenado.

Figura 53. Detección de YOLO reentrenado. Detección de ceda el paso y no detección de señal de stop.

4.7. Prueba 7. GPU y SSD Mobilenet reentrenado.

En esta prueba se hace uso del modelo congelado obtenido en el re-entrenamiento únicamente de la clase *señal de stop*, como se mencionaba en el apartado 3.2, dado que el entrenamiento llevado a cabo con las dos clases no llegó a converger y por tanto la detección quedaba comprometida.

Al igual que sucedía en la prueba 3 en comparación con la prueba 5, el número de detecciones es notablemente superior en SSD Mobilenet que en Yolo, esto es achacable principalmente al número de frames que obtenemos con SSD Mobilenet, notablemente superior al obtenido con Yolo.

Clase detectada	Nº Detecciones	% Confianza
Señal de stop	1649	92,17

Tabla 8. Clases detectadas en la Prueba 7.

Como se puede comprobar en la tabla 8, obtenemos 1649 detecciones con un porcentaje de confianza promedio de 92,17 %. Pese a obtener un porcentaje de confianza tan bueno, en el video de la imagen procesada podemos comprobar cómo aparece un gran número de falsos positivos, considerablemente superiores a los obtenidos en la prueba 2.

En lo que respecta al número de fps obtenidos en el tópico */frame_box*, comprobamos que los resultados de esta prueba mejoran substancialmente los de

la prueba 6, oscilando los valores obtenidos entre 13,9 y 15,5 fps frente al rango que presentaba Yolo reentrenado (entre 4,6 y 4,85 fps). Si comparamos el número de fps obtenidos en esta prueba con los resultados de la prueba 3, en la que obteníamos valores alrededor de 8,6, vemos que la mejoría es notable. La figura 54 muestra el gráfico obtenido de los valores filtrados de los fps que transporta el tópico con la imagen procesada por la red neuronal.

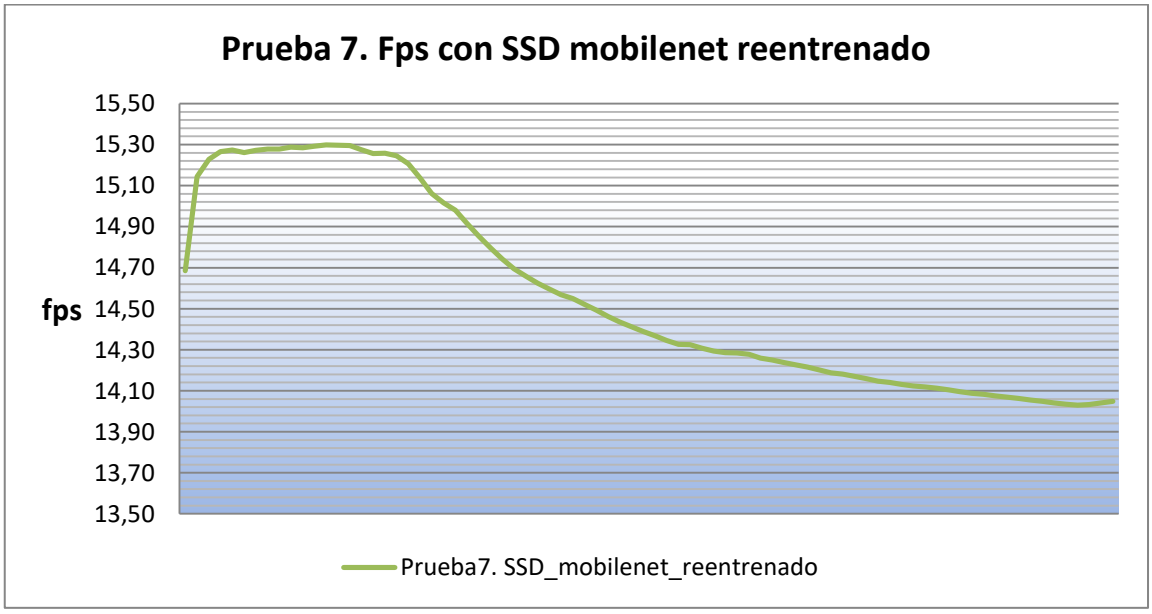


Figura 54. Fps obtenidos en la prueba 7.

Con respecto al rendimiento de la GPU, en la figura 55 vemos que el porcentaje de utilización es notablemente inferior al resultado obtenido en la prueba 6 y superior al obtenido en la prueba 3.

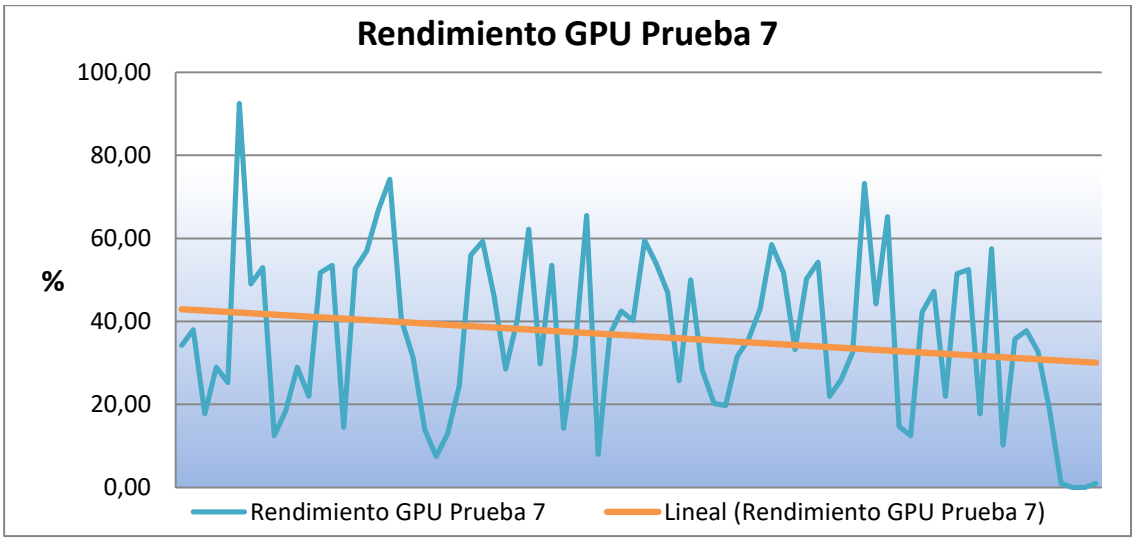


Figura 55 Rendimiento GPU Prueba 7.

De forma general podemos decir que el rendimiento de las CPUs es substancialmente superior que el que presenta YOLO reentrenado y similar al que obteníamos en la prueba 3.

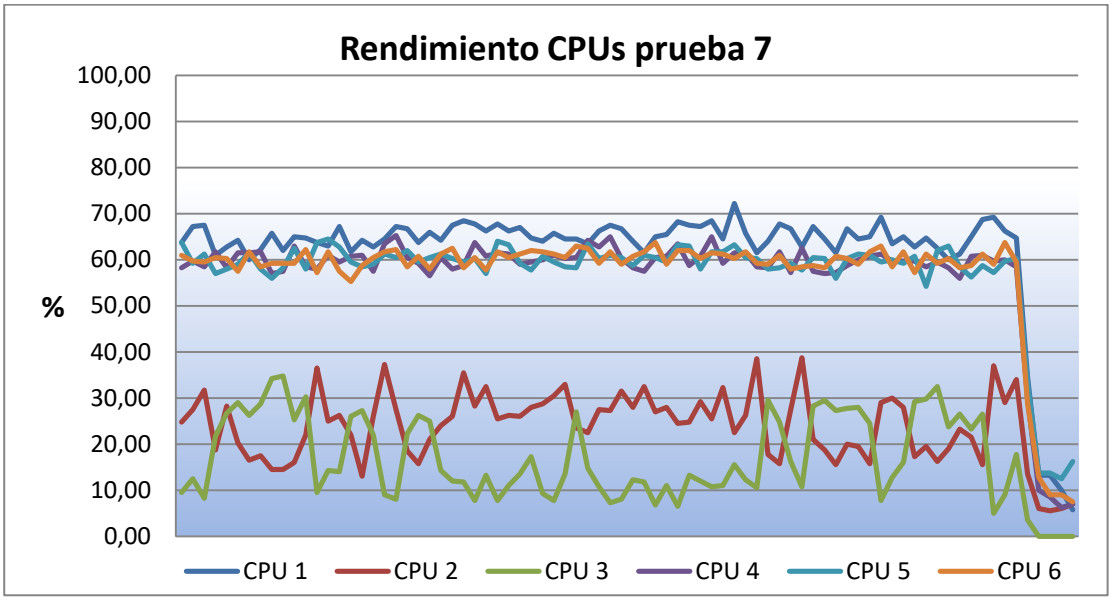


Figura 56. Rendimiento CPUs Prueba 7.

Desde un punto de vista cualitativo, vemos que el video obtenido tras el procesado de la imagen con la red neuronal es notablemente más fluido, presentando menos desfases y retrasos. Con respecto a la detección, vemos que hay un gran número de detección de falsos positivos, aunque por otro lado hay más persistencia en la detección de la señales de stop que aparecen en la escena.

La figura 57 muestra la detección de un falso positivo de señal de stop, ubicado en la rueda de la bicicleta.

Figura 57. Detección de falso positivo en la prueba 7.

La imagen de la izquierda de la figura 58 muestra que, pese a que la imagen de la presenta muchos brillos, el sistema la detecta la señal correctamente.

Figura 58. Detección de señal de stop en la prueba 7.

4.8. Prueba 8. GPU y Yolo reentrenado a tiempo real.

Como se cita al principio de este capítulo, se decide realizar dos últimas pruebas en las que no se trabaja sobre video, sino con imágenes tomadas a tiempo real, con el fin de plasmar los mejores resultados que podemos obtener con el sistema planteado. Los resultados de las principales variables, como sucedía en las otras pruebas, se capturan en archivos de registro y asimismo se captura el video gracias al hardware descrito en el Anexo I. Estas pruebas se llevan a cabo en interiores. Hemos de tener en cuenta que los recorridos realizados por el dron serán distintos para la prueba 8 y la 9.

La tabla 9 muestra las detecciones para cada una de las dos clases obtenidas tras llevar a cabo la prueba.

Clase detectada	Nº Detecciones	% Confianza
Señal ceda el paso	462	66,02
Señal de stop	74	66,22
Detecciones	**536**	**66,12**

Tabla 9. Clases detectadas en la Prueba 8.

Vemos que el sistema detecta 536 veces, abundando más la detección de la señal de ceda el paso que la de stop. El porcentaje promedio de confianza de detección ronda el 66%, significativamente inferior al obtenido en la prueba 6 y 7.

El número de fps obtenidos durante la prueba 8 es significativamente superior al de la prueba 6, como puede observarse en la figura 59. De esta forma, aparentemente, la publicación del tópico */frame* a partir del video procesado en el PC Virtual 1 parece ralentizar en cierta medida la detección, aunque aparentemente el número de fps en el tópico ya en la Jetson TX2 es de 30.

Así pues, este es el mejor resultado que se ha obtenido en lo que respecta a Yolo ROS, un promedio de **6,65 fps**.

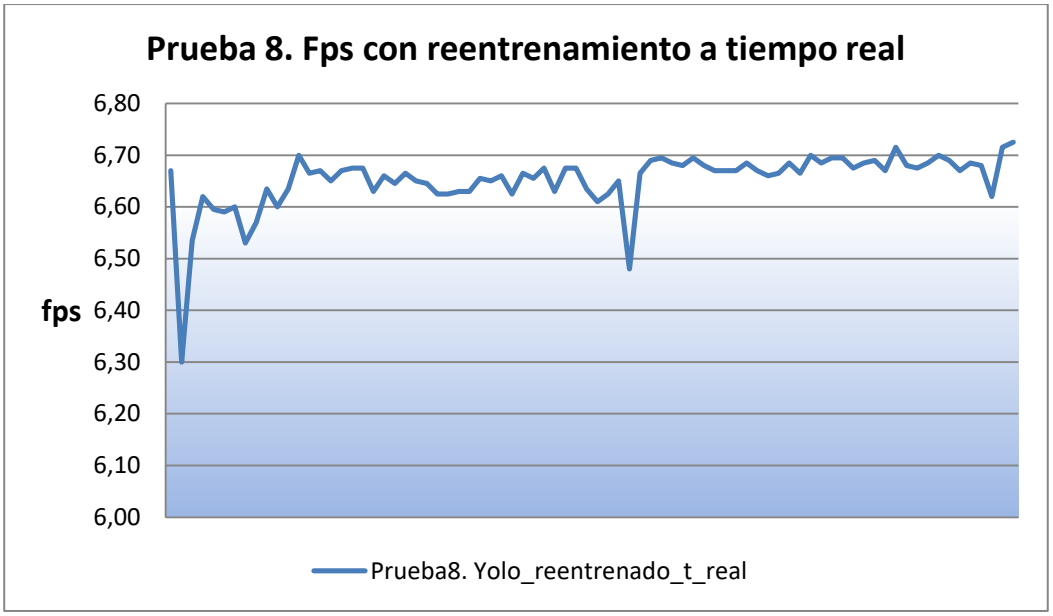


Figura 59. Fps obtenidos en la prueba 8.

En lo que respecta al rendimiento de la GPU, vemos que el porcentaje de uso aumenta en gran medida, estando el promedio muy próximo al 80 %. En el caso del porcentaje de uso de las CPU se observa un notable descenso en su porcentaje de uso. Parece lógico pensar que aumenta el uso de la GPU en detrimento del uso de las CPUs. Las figuras 60 y 61 muestran el rendimiento de hardware para la prueba 8.

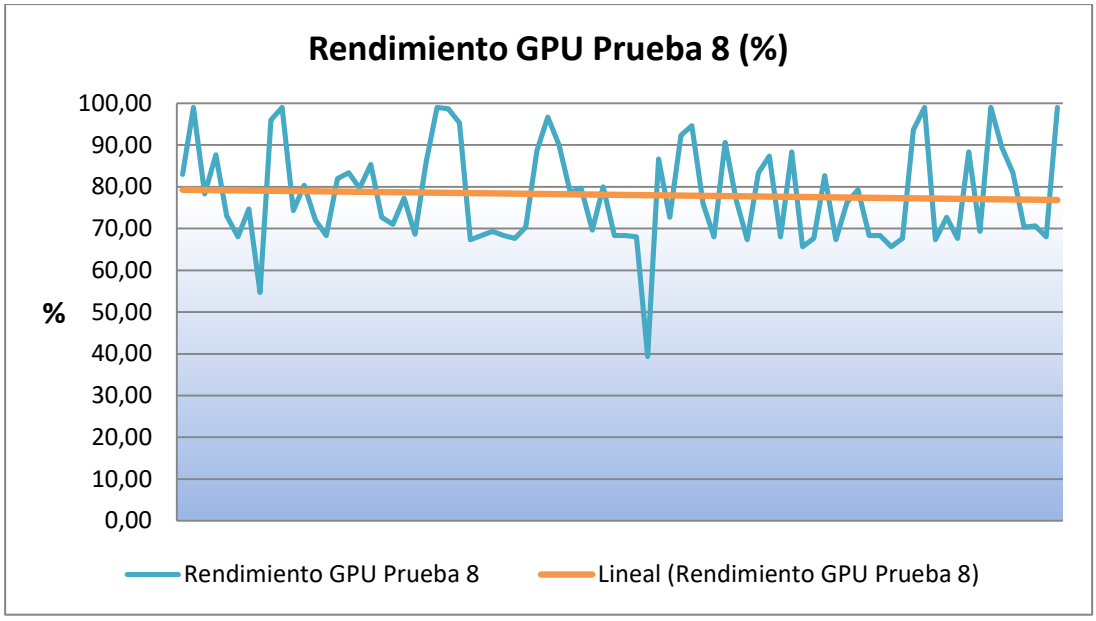


Figura 60. Rendimiento GPU Prueba 8.

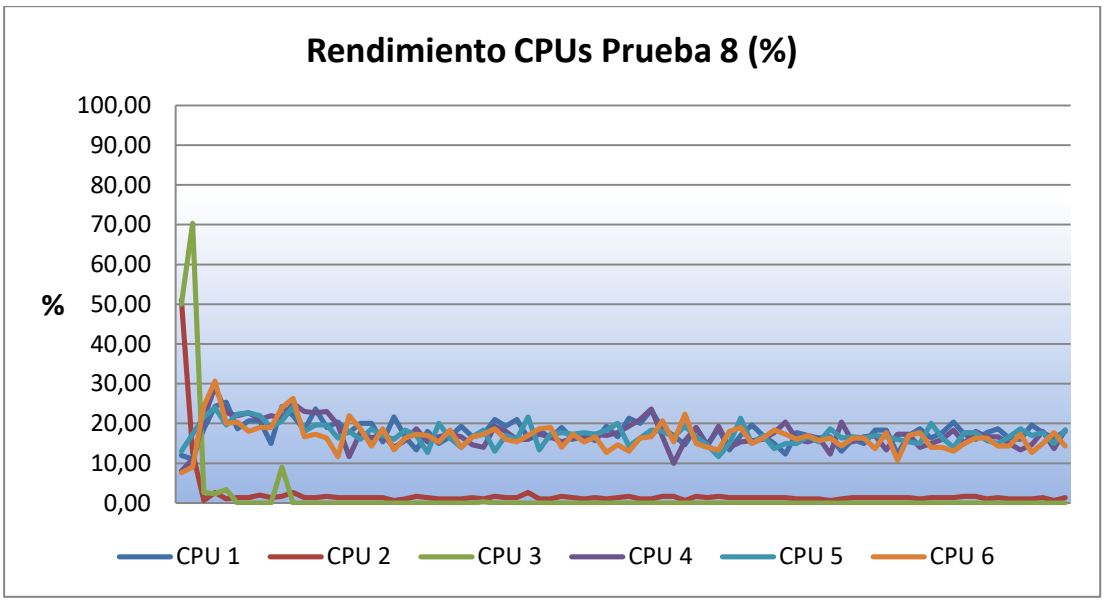


Figura 61. Rendimiento CPUs Prueba 8.

No se observan falsos positivos, pero sí vemos que la detección podría mejorar, ya que hay momentos en que las señales de stop y ceda el paso aparecen claramente en escena y sin embargo no son detectadas, como puede comprobarse en la imagen de la izquierda de la figura 63. La posición de las cajas también parece mejorable, presentando en ocasiones desfases con respecto a la posición del objeto a detectar, como se puede observar en la imagen de la derecha de la figura 62.

Figura 62. Detección de señal de ceda el paso en la prueba 8.

Figura 63. Detección de señal de ceda stop en la prueba 8.

4.9. Prueba 9.GPU y SSD Mobilenet re-entrenado a tiempo real.

Como se comenta en el apartado 3.2.2.2, se consigue reentrenar ssd_mobilenet_v1_coco con éxito en la detección únicamente con la clase *señal de stop*. La tabla 10 recoge el número de detecciones que logramos durante el desarrollo de la prueba 9.

Clase detectada	Nº Detecciones	% Confianza
Señal de stop	6001	99,52

Tabla 10. Clases detectadas en la Prueba 9.

Vemos en la tabla 10 que logramos unas 6000 detecciones con un porcentaje de confianza del 99,52 %. Pese a obtener un alto promedio de porcentaje de confianza en la detección, vemos cómo el sistema detecta numerosos falsos positivos, probablemente debido a que no hemos utilizado imágenes de evaluación para el reentrenamiento de la red neuronal. Este alto número de detecciones se debe a la

cantidad de fps logrados con el sistema utilizado en esta prueba. Como podemos observar en la figura 64, el número de fps en todo caso supera los **20 fps**, con picos que llegan a superar los **21**.

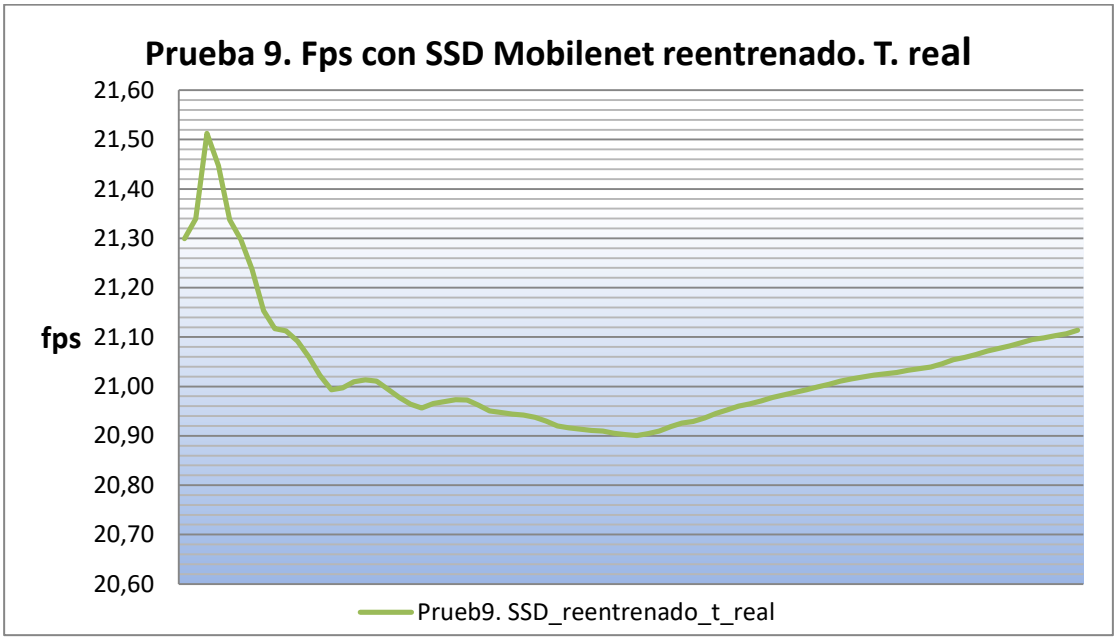


Figura 64. Fps obtenidos en la prueba 9.

En lo que respecta al rendimiento del hardware, vemos cómo aumenta el porcentaje de uso de la GPU y se reduce el de las CPUs si lo comparamos con la prueba 7,. El rendimiento de la GPU y CPUs se puede observar en las figuras 65 y 66.

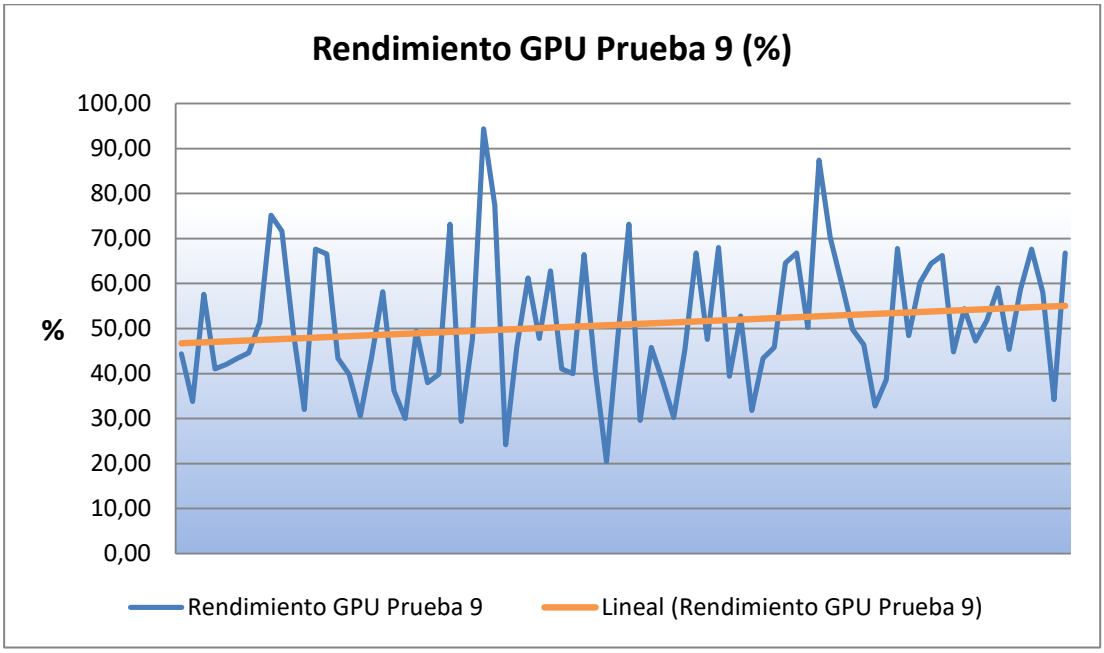


Figura 65. Rendimiento GPU Prueba 9.

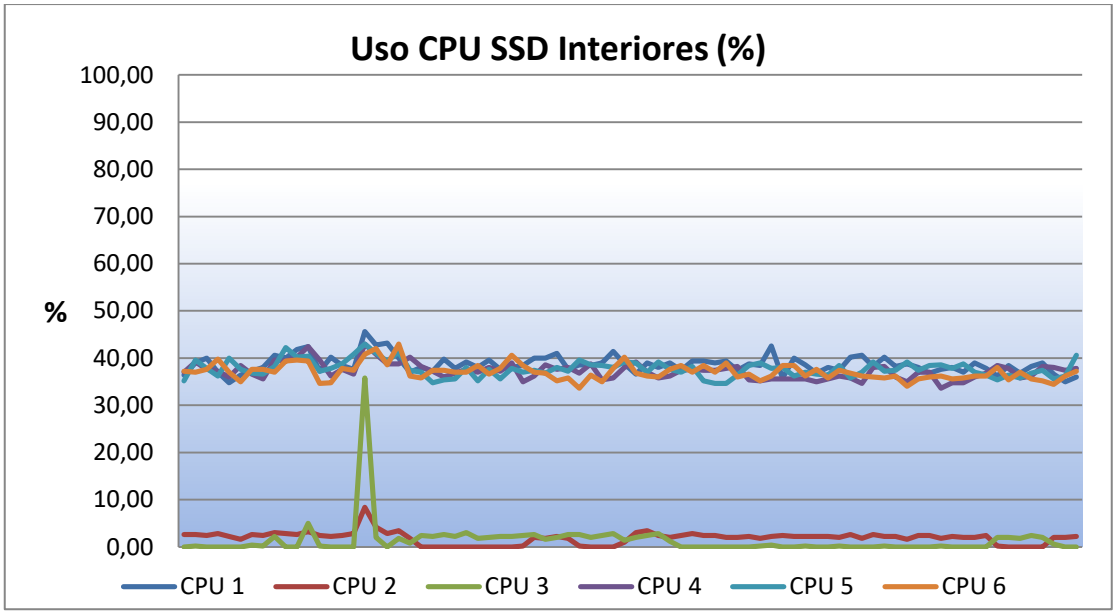


Figura 66. Rendimiento CPUs Prueba 9.

Como sucedía en la prueba 7, observamos que se detectan numerosos falsos positivos a lo largo de la prueba, como se puede observar en la figura 67.

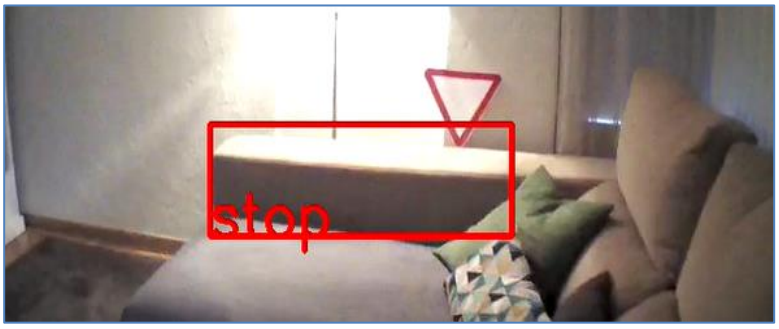


Figura 67. Detección de falso positivo en la prueba 9.

Sin embargo, observamos que el sistema tiene una muy buena reacción en la detección temprana de la señal de stop, como se demuestra en la figura 68, en la que se ve cómo el sistema detecta la señal cuando aún sólo se observa la mitad de la misma.

Figura 68. Detalle de la detección de señal de stop en la prueba 9.

4.10. Análisis comparativo de resultados del rendimiento de la detección.

Una vez analizados los resultados individualmente, a continuación se presenta un resumen comparativo que aporta una perspectiva global de cara a extraer conclusiones. El análisis se centra en los fps obtenidos, así como en el rendimiento de la Jetson TX2.

En primer lugar estudiamos los fps obtenidos de forma agrupada y en segundo lugar el rendimiento del hardware de forma comparativa para tener una mejor visión del conjunto. La figura 69 muestra un gráfico con el promedio de fps obtenido en el tópico que transporta la imagen ya procesada por la red neuronal en cada prueba realizada.

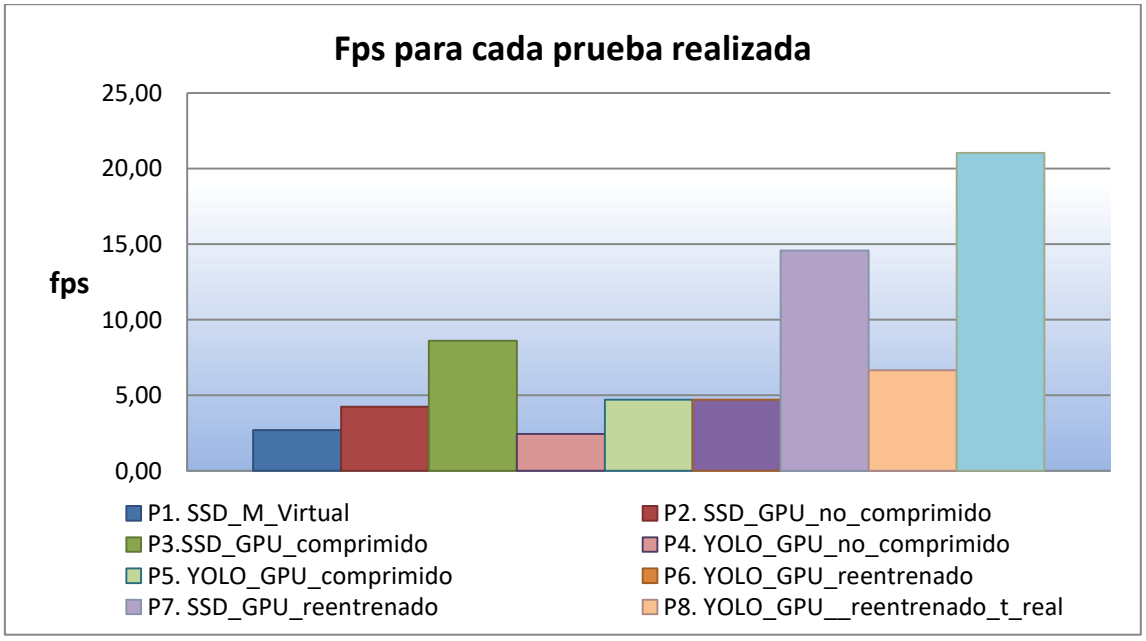


Figura 69. Comparativa de fps obtenidos en la imagen procesada de cada prueba.

La figura 70 muestra el promedio de los fps obtenidos en las pruebas llevadas a cabo con el modelo SSD. Se puede observar una gran mejora de los resultados en la transición de la prueba 1 a la 9.

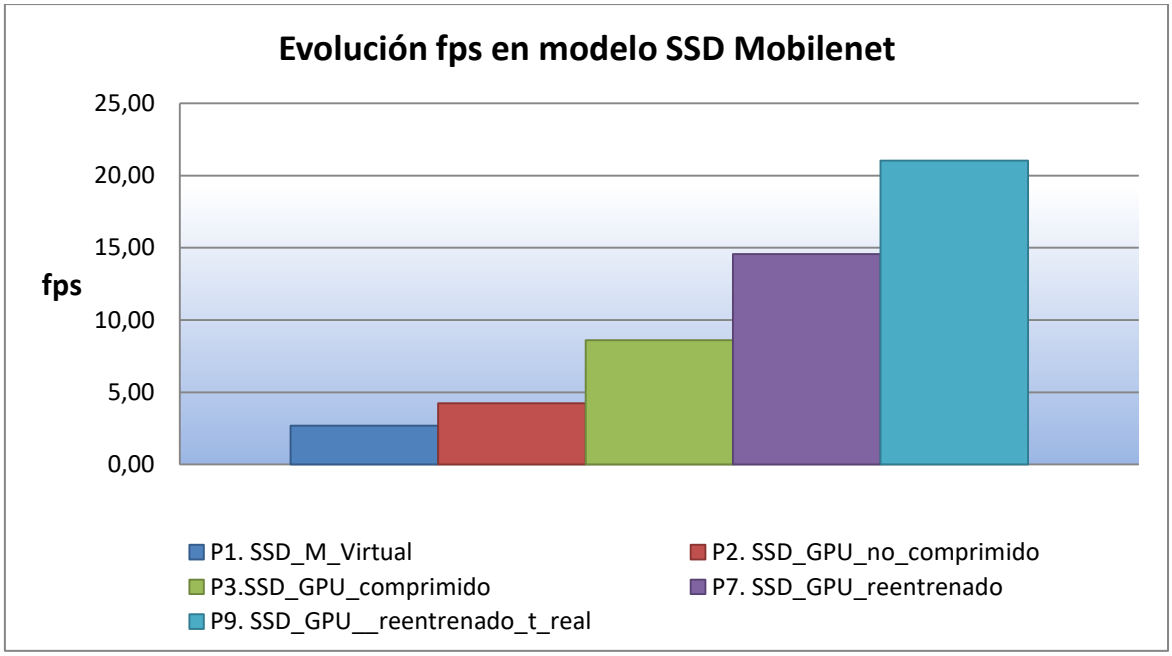


Figura 70. Evolución en los fps obtenidos tras el procesado con ssd_mobilenet v1

La figura 71 muestra la evolución en los fps obtenidos en el procesado con el modelo YOLO. Pese a que también se observa una mejora en la transición de la prueba 4 a la 8, no es tan acusada como loa que se observa en el modelo SSD Mobilenet (figura 70).

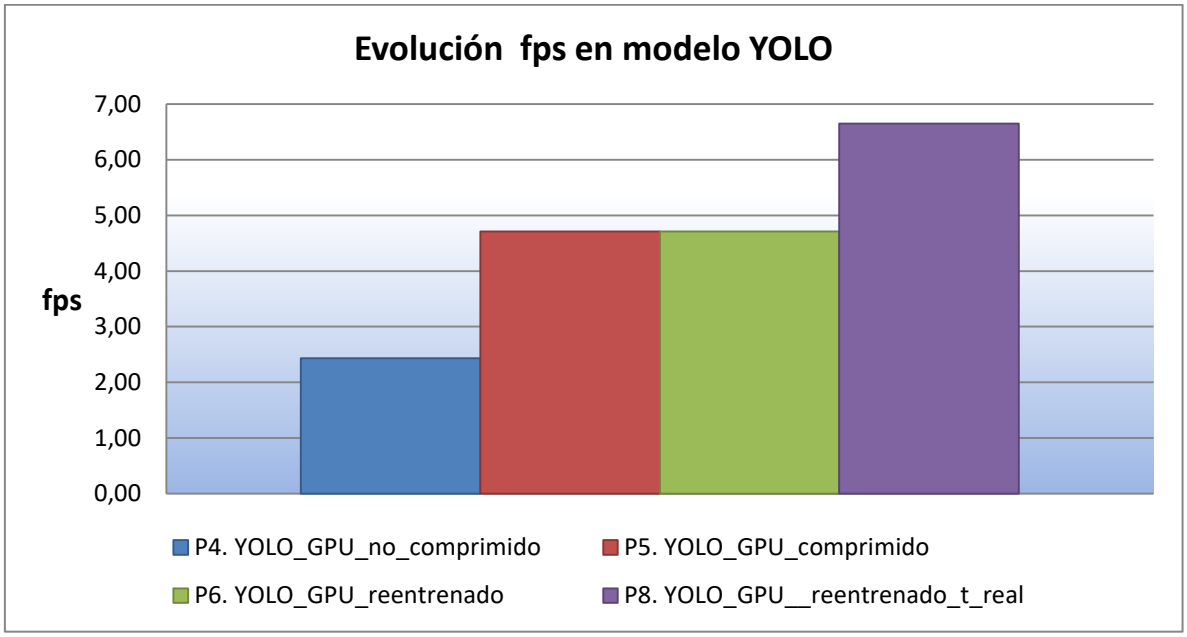


Figura 71. Evolución en los fps obtenidos tras el procesado con YOLO.

A continuación analizamos el rendimiento de la Jetson TX2. La figura 72 que vemos a continuación muestra un resumen global.

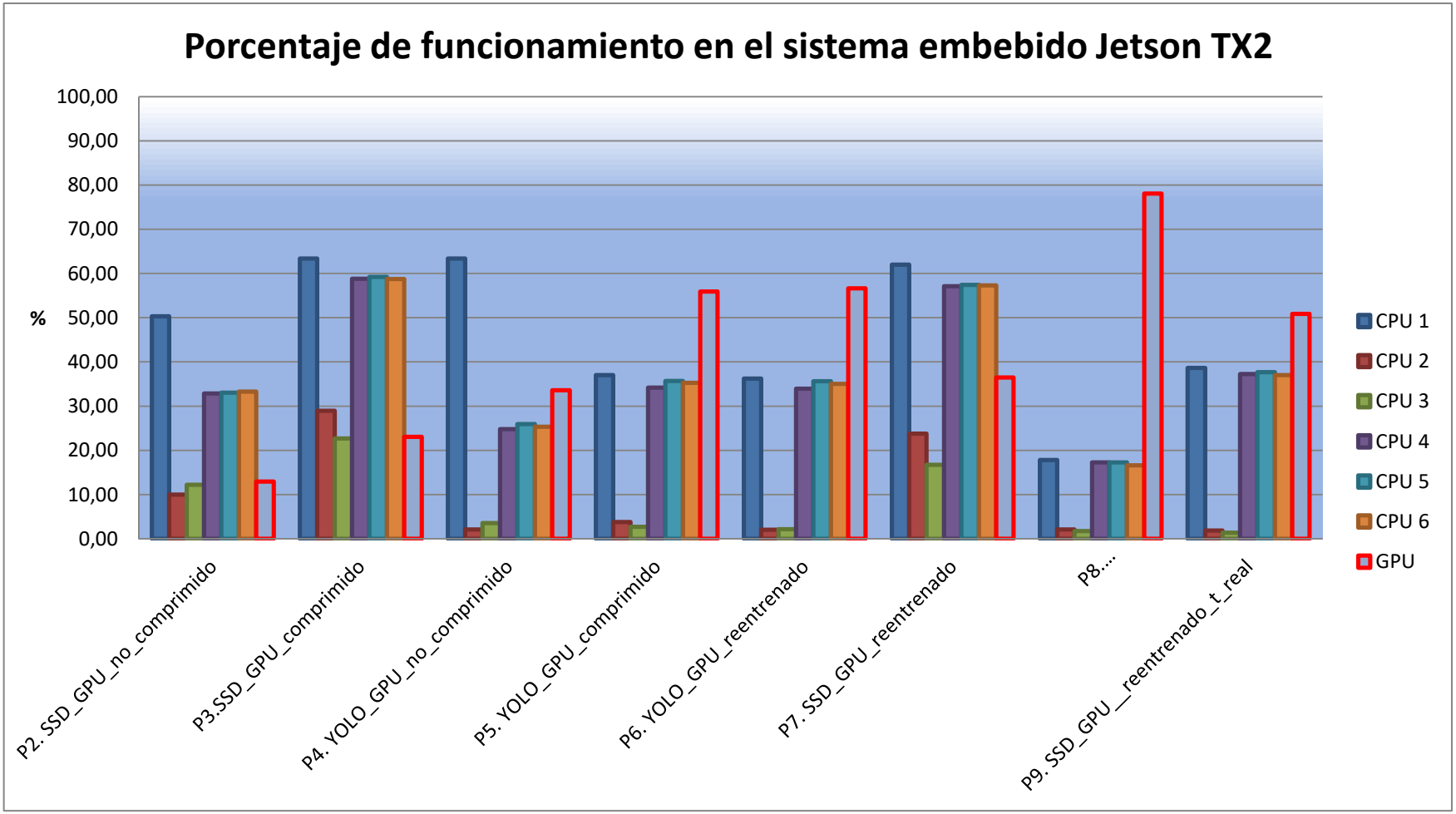


Figura 72. Rendimiento de hardware promedio en todas las pruebas realizadas.

La figura 73 muestra los resultados en el uso de hardware para las pruebas que usan imágenes sin comprimir.

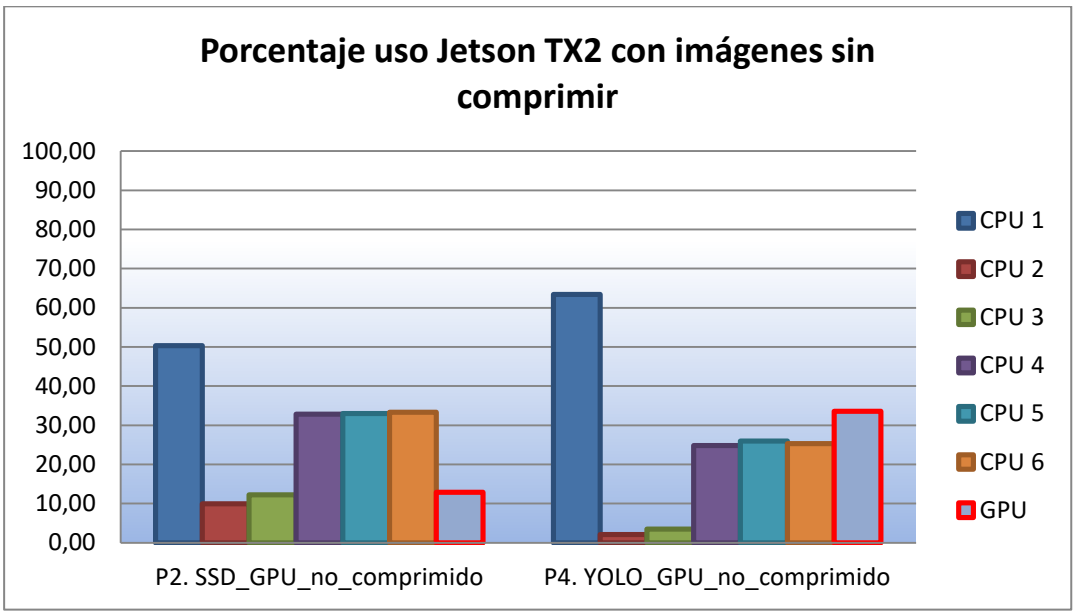


Figura 73. Rendimiento del hardware en el procesado de imágenes sin comprimir.

La figura 74 muestra el gráfico obtenido para el uso de hardware en las pruebas realizadas con imágenes comprimidas y los modelos originales *ssd_mobilenet_v1_coco*y YOLO respectivamente.

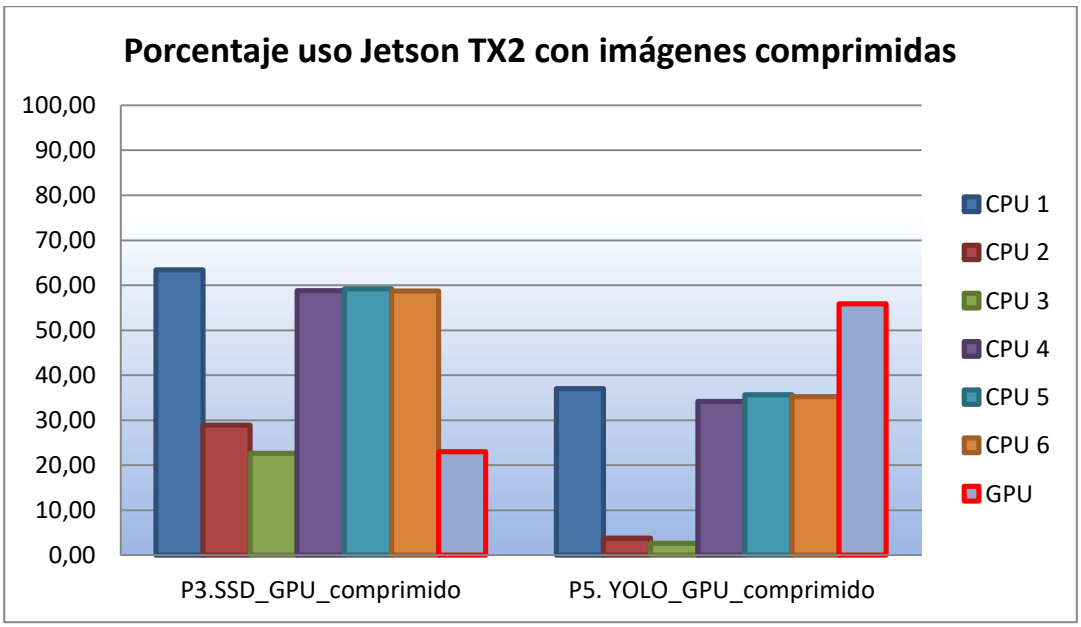


Figura 74. Rendimiento del hardware en el procesado de imágenes comprimidas.

La figura 75 muestra los resultados obtenidos para el modelo SSD en lo que respecta a rendimiento de hardware.

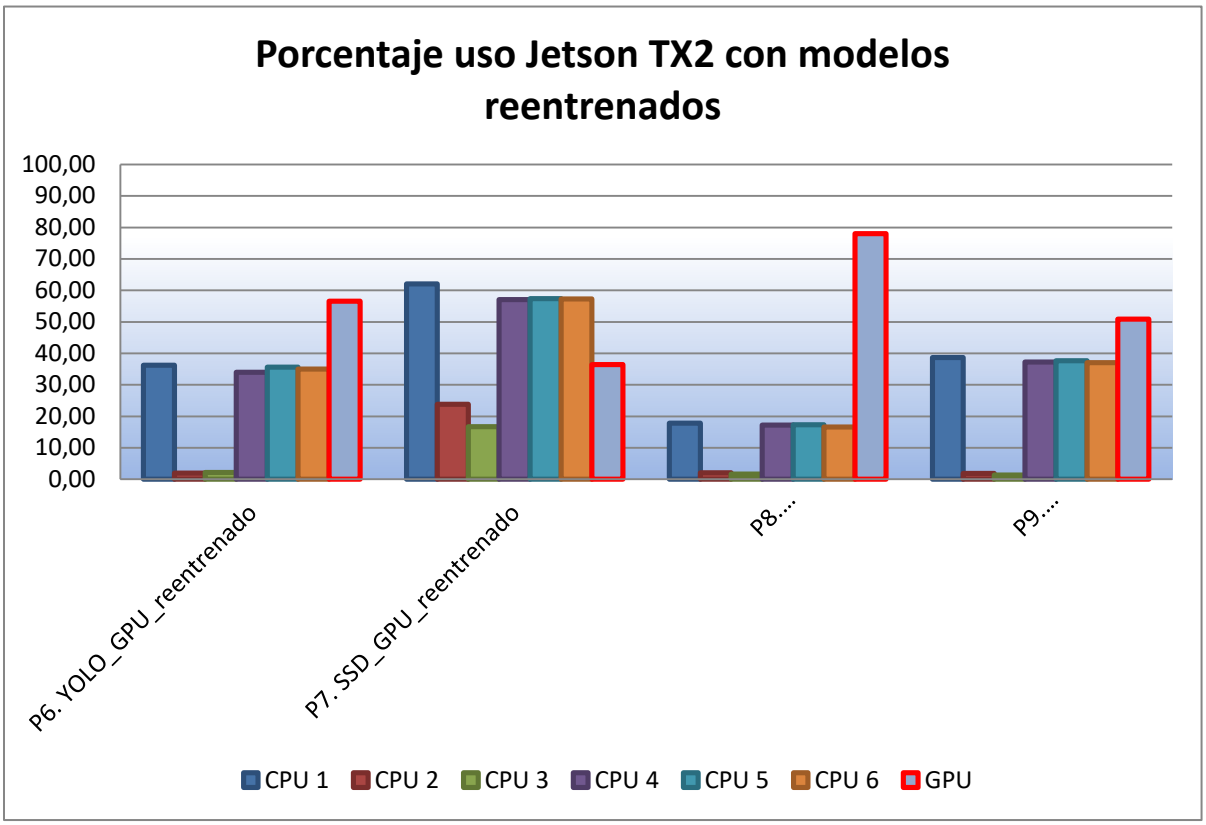


Figura 75. Rendimiento del hardware en el procesado de imágenes comprimidas

En lo que respecta al número de fps, se observa, que en general, los resultados obtenidos con el modelo ssd_mobilenet_v1_coco son mejores que los obtenidos con YOLO. Además, vemos cómo, el sistema mejora notablemente la velocidad de proceso a lo largo de las pruebas realizadas con ambos modelos. En primera instancia gracias al uso de la GPU y en segundo lugar y de forma muy significativa gracias a la compresión de imágenes transportadas en el tópico */frame*, que parece clave para mejorar el la velocidad del sistema..

En lo que respecta al rendimiento del sistema embebido Jetson TX2, vemos que, de forma general, YOLO usa más la GPU y menos las CPUs, justo al contrario que ssd_mobilenet_v1_coco. Además vemos como el porcentaje de uso tanto de CPU como de GPU es inferior en caso del uso de imágenes sin comprimir. Por último, destacamos que en la ejecución de los modelos reentrenados se usa más la GPU y menos las CPUs si lo comparamos con los modelos estándar.

5. Capítulo 5. Conclusiones y Trabajo futuro.

En lo que respecta al primer objetivo planteado en el capítulo 1, la consecución de un sistema de pilotaje remoto que nos permita obtener asimismo imágenes del dron para procesarlas a tiempo real, podemos concluir que se ha logrado parcialmente el mismo, ya que, pese a haber conseguido integrar el sistema propuesto para el control del dron, se comprueba que la estabilidad que conseguimos en pilotaje del UAV no es la óptima para el espacio en el que se desarrollan las pruebas, necesitando mucho terreno para controlarlo remotamente de forma segura.

En cuanto al estudio de las dos alternativas a la detección de objetos, se ha logrado integrar y hacer funcionar los dos sistemas objeto de análisis, YOLO y SSD Mobilenet. De hecho, se consigue una gran mejora en la velocidad de proceso y una notable optimización desde el planteamiento inicial hasta el planteamiento final, gracias fundamentalmente al uso de un hardware de alto rendimiento como es el sistema embebido Jetson TX2 y a lo que ha resultado la clave para la optimización de los dos sistemas, la compresión de imágenes en ROS.

Se ha logrado, asimismo, reentrenar las redes en los dos sistemas planteados con imágenes de señales de tráfico (ceda el paso y de stop) procedentes de distintos conjuntos de datos. En el caso del re-entrenamiento con TensorFlow, pese a que el proceso para las dos clases finaliza sin ningún tipo de error, al probarlo, se observa que no detecta ninguno de los dos objetos con los que se ha reentrenado, por tanto el entrenamiento no llegó a converger, comprobando así la afirmación que se recoge en el apartado 2, en el que se menciona que el entrenamiento de SSD se plantea como un reto único y donde también se comenta que un reentrenamiento puede que no converja debido a un insuficiente número de muestras o bien a muestras no representativas (en nuestro caso, es posible que se deba a un número insuficiente de muestras, concretamente de la clase *señal de ceda el pa*so). Así pues, en el caso del re-entrenamiento con TensorFlow y el modelo SSD Mobilenet sólo conseguimos el reentrenamiento para una de las clases, la señal de stop.

En general podemos afirmar que en las pruebas llevadas a cabo y como parece lógico, tenemos más detecciones cuanto mayor es la velocidad de proceso. De esta forma, en SSD Mobilenet obteníamos 313 detecciones a unos 2,7 fps en la primera prueba, llegando a 1940 con el mismo modelo a 8,6 fps. En YOLO tenemos un comportamiento similar, pasando de 264 detecciones a unos 2,2 fps en la prueba 4 a unas 609 detecciones a unos 4,7 fps en la prueba 5.

En el caso de los modelos re-entrenados, vemos cómo en SSD conseguimos una mejora significativa en el modelo re-entrenado con respecto al modelo original. En YOLO, pese a que existe una mejora en la velocidad de procesado, no se produce un salto tan grande.

Desde un punto de vista cualitativo, resaltamos que a menos fps, YOLO presenta una imagen de salida ya detectada aparentemente más fluida que la que presenta SSD.

En lo que respecta al porcentaje de confianza de los sistemas estudiados, vemos que existe una moderada cantidad de falsos positivos, especialmente en SSD, tanto en el modelo original como en el reentrenado. YOLO proporciona menos porcentaje de falsos positivos en las pruebas realizadas, presentando aparentemente, mayor confianza en la detección, pese a estar configurado para detectar con un índice de confianza de 0,5 frente al 0,7 de SSD. También comprobamos que obtenemos mayor porcentaje de confianza a más fps obtenidos en la imagen ya procesada por la red neuronal, así, en la prueba 1, a unos 2,7 fps obtenemos un porcentaje promedio de 54,41%, mientras que en la prueba 3, a unos 8,6 fps obtenemos un promedio de 65,58%. En YOLO el salto no es tan grande, pasamos de un 64,47% a unos 2,2 fps en la prueba 4 a un 68,28% a unos 4,7 fps en la prueba 5.

El rendimiento de la GPU es mayor en YOLO que en SSD, así, en las dos últimas pruebas, que son las de resultados a tiempo real, obtenemos un promedio de porcentaje de uso de la GPU próximo al 80 % para YOLO a 6,65 fps frente a un 50 % aproximado de uso para la obtención de unos 21 fps en SSD. En el uso de las CPUs sucede exactamente al contrario, siendo el porcentaje de uso de las mismas significativamente inferior en YOLO que en SSD Mobilenet. Esto se debe muy probablemente a la arquitectura y a la gestión de CUDA en cada uno de los sistemas propuestos.

Podemos concluir diciendo que hemos conseguido mayor velocidad de proceso con SSD Mobilenet llegando a superar los 21 fps en la prueba número nueve, siendo por tanto el sistema que más se aproxima a tiempo real de los analizados en este trabajo. Sin embargo, la confianza de la detección tiene amplio margen de mejora. Esta optimización se podría conseguir sin ninguna duda llevando a cabo sucesivos re-entrenamientos hasta encontrar la solución óptima.

De cara a posibles trabajos futuros, cabe mencionar en primer lugar la búsqueda de un dron que presentara una mayor estabilidad que el utilizado durante este proyecto y que llevara integrado el sistema embebido, de cara a optimizar aún más el proceso y aproximarnos a tiempo real en mayor medida. Si se continuara trabajando con el mismo dron, habría que trabajar en la línea de búsqueda de auto-estabilización automática del mismo con el driver propuesto.

En lo que respecta a la detección propiamente dicha, no parece que YOLO v1 pueda dar mucho más de sí, así que se podría plantear seguir investigando mejoras con YOLO v2 o con YOLO v3 integrado en ROS.

Si se optara por TensorFlow, que parece lo lógico, vista la multitud de opciones que ofrece y vistos los resultados con tan sólo un promedio aproximado del 50 % de uso de la GPU, se podría inicialmente seguir investigando en el re-entrenamiento del modelo SSD basado en Mobilenets. También se podría tratar de optimizar CUDA para su aplicación al sistema.

Podría, finalmente, enfocarse un posible trabajo futuro basado en vuelo autónomo combinado con mapas locales y coordenadas GPS, haciendo uso de aprendizaje profundo y redes neuronales convolucionales aplicado a distintas tareas dentro del campo de la agricultura de precisión.

REFERENCIAS

1. ARDRONE AUTONOMY (2018). Disponible on-line: https://ardrone autonomy.readthedocs.io/en/latest/ (accedido Mayo 2018).

2. ARDRONE AUTONOMY GPS (2018). Disponible on-line: https://ardrone-autonomy.readthedocs.io/en/gps-waypoint/gps.html (accedido Mayo 2018).

3. COCO DATASET (2018). Disponible on-line: http://cocodataset.org/#home (accedido Agosto 2018).

4. CUDA (2018). Disponible on-line: https://www.nvidia.es/object/cuda-parallel-computing-es.html (accedido Julio 2018).

5. Dai, J., Li, Y., He, K., Sun, J. (2016). R-FCN: Object Detection via Region-based Fully Convolutional Networks. arXiv:1605.06409v2 [cs.CV].

6. Dertat, A. (2017). Applied Deep Learning - Part 4: Convolutional Neural Networks. Disponible on-line: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2 (accedido Mayo 2018).

7. Forson, E. (2017). Understanding SSD Multibox-Real Time Object Detection In Deep Learning. Disponible on-line: https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab (accedido Agosto 2018).

8. Gigioli, P. (2017-2018). YOLO integration with ROS for real-time object detection. Disponible on-line: https://github.com/pgigioli/darknet_ros (accedido Junio 2018).

9. Girshick, R., Donahue, J., Darrell, T., Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. arXiv:1311.2524 [cs.CV].

10. Girshick, R. (2015) Fast R-CNN. arXiv:1504.08083v2 [cs.CV].

11. Guanghan, N. (2015) Start Training YOLO with Our Own Data. Disponible on-line: http://guanghan.info/blog/en/my-works/train-yolo/ (accedido Mayo 2018).

12. Haykin, S. (1999) Neural networks a comprehensive foundation. IEEE Press, New York, USA.

13. Hui, J. (2018). Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3. Disponible on-line:https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088 (accedido Agosto 2018).

14. INSTALACIÓN OPENCV (2018). Disponible on-line: https://www.pyimagesearch.com/2016/10/24/ubuntu-16-04-how-to-install-opencv/ (accedido en Mayo 2018).

15. INSTALACIÓN TENSORFLOW (2018). Disponible on-line: https://www.TensorFlow.org/install/install_sources (accedido Mayo 2018).

16. JETSON TX2 (2018). Disponible on-line: https://www.nvidia.es/autonomous-machines/embedded-systems-dev-kits-modules/ (accedido Agosto 2018).

17. JETSONHACKS (2016-2018). Install Robot Operating System (ROS) on NVIDIA Jetson TX2. Disponible on-line: https://github.com/jetsonhacks/installROSTX2(accedido Junio 2018).

18. Joyce, X. (2017). Deep Learning for Object Detection: A Comprehensive Review. Disponible on-line: https://towardsdatascience.com/deep-learning-for-object-detection-a-comprehensive-review-73930816d8d9 (accedido Agosto 2018).

19. Lee, P. (2018). TensorFlow for NVIDIA Jetson. Disponible on-line: https://github.com/peterlee0127/TensorFlow-nvJetson(accedido Julio 2018)

20. Levy, S. (2014, 2018). Auto-Pilot the Parrot AR.Drone from Python (or Matlab or C). Disponible on-line: https://github.com/simondlevy/ARDroneAutoPylot (Accedido Junio 2018).

21. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y., Berg, A.C. (2016). SSD: Single Shot MultiBox Detector. arXiv:1512.02325v5 [cs.CV].

22. MATLAB (2018). The MathWorks. Disponible on-line: https://es.mathworks.com/products/matlab.html (accedido Mayo 2018)

23. Møgelmose, A., Trivedi, M.M., Moeslund, T.B. (2012). Vision based Traffic Sign Detection and Analysis for Intelligent Driver Assistance Systems: Perspectives and Survey. IEEE Transactions on Intelligent Transportation Systems, 13(4), 1484-1497.

24. Murugavel, M. (2018). Yolo-Annotation Tool. Disponible on-line: https://github.com/ManivannanMurugavel/YOLO-Annotation-Tool (accedido Julio 2018).

25. Nielsen, M. (2017). Neural Networks and Deep Learning. Disponible on-line: http://neuralnetworksanddeeplearning.com/ (accedido Agosto 2018).

26. NUMPY (2018). Disponible on-line: http://www.numpy.org/ (Accedido Agosto 2018).

27. NVPMODEL JETSON (2018). Disponible on-line: https://www.jetsonhacks.com/2017/03/25/nvpmodel-nvidia-jetson-tx2-development-kit/ (accedido Julio 2018).

28. OPEN CV (2018). Disponible on line: https://opencv.org/ (accedido Julio 2018).

29. Pajares, G. (2015). Overview and Current Status of Remote Sensing Applications Based on Unmanned Aerial Vehicles (UAVs). Photogrammetric Engineering and Remote Sensing, 81(4), 281–329.

30. Pajares, G. y de la Cruz, J.M. (2007). Visión por computador: imágenes digitales y aplicaciones, RA-MA, Madrid.

31. PASCAL VOC (2018). Disponible on-line en: http://host.robots.ox.ac.uk/pascal/VOC/ (accedido Agosto 2018).

32. PESOS YOLO (2018). Disponible on-line: http://pjreddie.com/media/files/darknet19_448.conv.23 (accedido Agosto 2018).

33. Redmon, J., Divvala, S., Girshick, R., Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. arXiv:1506.02640v5[cs.CV].

34. Redmon, J. C. (2012) Darknet (2013-2016): Open Source Neural Networks in C. Disponible on-line: https://pjreddie.com/darknet/ (accedido Agosto 2018).

35. Redmon, J., Farhadi, A. (2016). YOLO9000: Better, Faster, Stronger. arXiv:1612.08242 [cs.CV].

36. Redmon, J., Farhadi, A. (2018). YOLOv3: An Incremental Improvement. arXiv:1804.02767 [cs.CV].

37. Ren, S., He, K., Girshick, R., Sun, J. (2016). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. arXiv:1506.01497v3 [cs.CV].

38. ROS(2018). Disponible on-line: http://www.ros.org/ (accedido Junio 2018).

39. Solem, J. E. (2012). Programming Computer Vision with Python. Creative commons. O'Reilly Media, US. Disponible on-line: http://programmingcomputervision.com/downloads/ProgrammingComputerVision_CCdraft.pdf (accedido Junio 2018).

40. ROS CV_BRIDGE (2018). Disponible on-line: http://wiki.ros.org/cv_bridge (accedido Julio 2018).

41. Stang, D. (2017). Step by Step TensorFlow Object Detection API Tutorial. Disponible on-line: https://medium.com/@WuStangDan (accedido Mayo 2018).

42. Szegedy, C., Reed, S., Erhan, D., Anguelov, D., Ioffe, S. (2015). Scalable High Quality Object Detection. arXiv:1412.1441v3 [cs.CV].

43. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A. (2014). Going Deeper with Convolutions. arXiv:1409.4842v1 [cs.CV].

44. TENSORFLOW (2018). Disponible on-line: https://www.tensorflow.org/ (accedido Junio 2018).

45. TENSORFLOW MODEL ZOO (2018). Disponible on-line: https://github.com/TensorFlow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md (accedido Agosto 2018).

46. THE UAV (2018). Disponible on-line: https://www.theuav.com/(accedido Mayo 2018).

47. TRANSPORTE IMAGEN ROS (2018). Disponible on-line: http://wiki.ros.org/image_transport#Python_Usage (accedido Mayo 2018).

48. Vega, J. A., Dormido-Canto, S. (2010). Máquinas de Vectores Soporte. Aprendizaje Automático (Pajares, G. y de la Cruz, J.,M., Eds.). RA-MA, Madrid.

49. VIRTUAL BOX (2018). Disponible on-line: https://www.virtualbox.org/ (accedido Mayo 2018).

50. XENIAL XERUS (2018). Disponible on-line: http://releases.ubuntu.com/16.04/ (Accedido Mayo 2018)

ANEXO I. Hardware utilizado

Este anexo recoge un listado del hardware utilizado en el TFM, así como las características del mismo.

✓ **Drone AR2.0 de Matlab, con las siguientes características:**

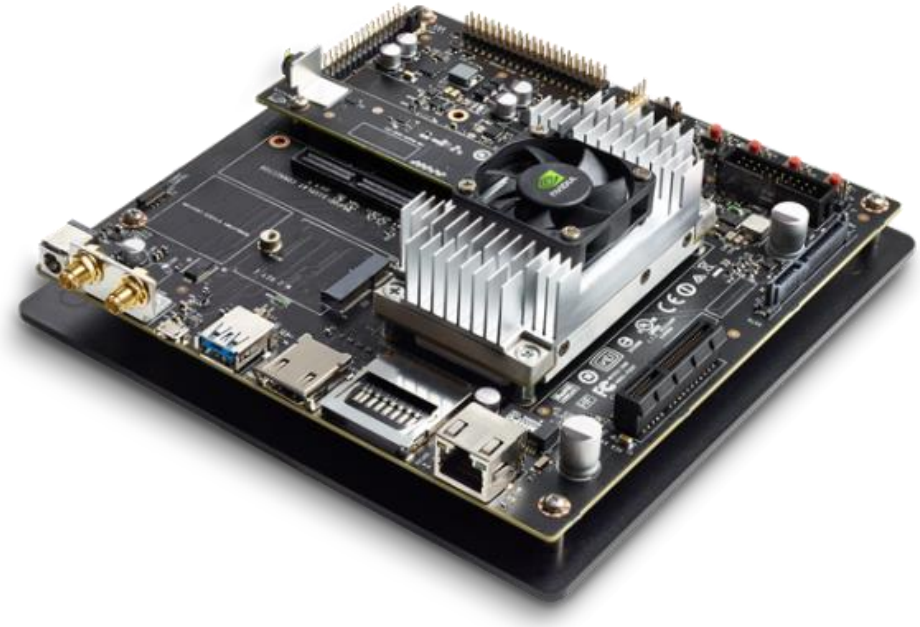
- Procesador ARM Cortex de 32 bits a 1 GHz.
- DSP TSM320DMC64x a 800 MHz
- Giroscopio de tres ejes ultra rápido a 2000 °/s
- Acelerómetro
- Magnetómetro: 3 ejes, precisión de 6°
- Sensor de presión: Precisión de ± 10 Pa
- Sensor de ultrasonidos
- Cámara frontal de 720p a 30fps.
- Objetivo: Objetivo gran angular: diagonal 92°
- Perfil de codificación básica: H264
- Formato fotos: JPEG
- Cámara inferior QVGA de 60fps.
- Conexión: WI-FI

✓ **Ordenador portátil Lenovo 50-80, con las siguientes características:**

- Procesador Intel ® Core™ i7 5500U CPU @ 2.4 Ghz.
- 16 Gb de Ram DDR3.
- Disco Duro SSD de 250 Gb sobre el que se instala el sistema operativo.
- Disco duro HD de 500Gb.

- ✓ **Kit de desarrollo Jetson TX2, con las siguientes características:**

Gráficos	256 núcleos NVIDIA Pascal™
CPU	HMP Dual Denver 2/2 MB L2 + Quad ARM® A57/2 MB L2
Video	Codificación 4K x 2K a 60 Hz (HEVC) Descodificación 4K x 2K a 60 Hz (12 bits)
Memoria	LPDDR4 de 8 GB y 128 bits 59,7 GB/s
Pantalla	2 interfaces DSI, 2 DP 1.2 / HDMI 2.0 / eDP 1.4
CSI	Hasta 6 cámaras (2 vías) CSI2 D-PHY 1.2 (2,5 Gbps/vía)
PCIE	Gen 2 | 1x4 + 1x1 OR 2x1 + 1x2
Almacenamiento De Datos	eMMC, SDIO, SATA de 32 GB
Otros	CAN, UART, SPI, I2C, I2S, GPIOs
USB	USB 3.0 + USB 2.0
Conectividad	1 Gigabit Ethernet, 802.11ac WLAN, Bluetooth
Mecánicas	50 mm x 87 mm (conector tarjeta-tarjeta compatible de 400 patillas)



✓ **Switch - TP-Link, 5 puertos, 10/100Mbps**

- Velocidad de transferencia 10/100 Mbps.
- Detección automática MDI/MDIX para evitar la necesidad de usar cables cruzados.
- 5 puertos Ethernet.

✓ **Punto de acceso**

- Banda de frecuencia: 2.4 GHz
- Velocidad de transferencia LAN: 10/100 Mbps.

✓ **Capturadora de video Corsair Elgato HD.**

Usaremos la capturadora de video para llevar a cabo grabaciones de las pruebas realizadas en la GPU. La capturadora nos permite grabar en alta definición.

- Formato de vídeo soportado: 1080i, 480i, 576p, 720p

ANEXO II. Preparación del entorno en el PC Virtual 1.

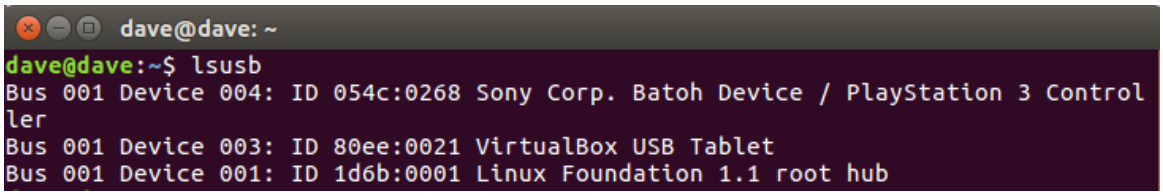
Este anexo recoge los pasos seguidos para la preparación y compilación del repositorio del que partimos *Levy, (2014, 2018)*. Para que el sistema se pueda utilizar en ROS, llevamos a cabo la modificación de tres archivos fundamentales:

- autopylot.makefile
- autopylot_python_agent.c
- autopylot_agent.py

A continuación vemos en detalle cómo se ha adaptado el repositorio para la integración en este trabajo.

- ***autopylot.makefile***

Una vez conectado el mando de PS3 a un puerto USB, comprobamos la ID del joystick gracias al comando lsusb.



Visualización de la ID del Joystick conectado a puerto USB

Como podemos comprobar, la ID del joystick es 054c:0268.

Modificamos por tanto el archivo autopylot.makefile con la siguiente línea:

GAMEPAD = GAMEPAD_PS3_ID=0X054C0268

No haría falta modificar ninguna línea más, se trabaja con Python por defecto.

- ***autopylot_python_agent.c***

Cuando se inicia este proyecto, el repositorio ARDroneAutoPylot no estaba preparado para funcionar con ROS. Tras llevar a cabo trabajo personal y colaborativo, se consigue integrar el sistema en ROS, creando un nodo llamado dron_camera, que a posteriori será el que publique el tópico con la imagen del dron.

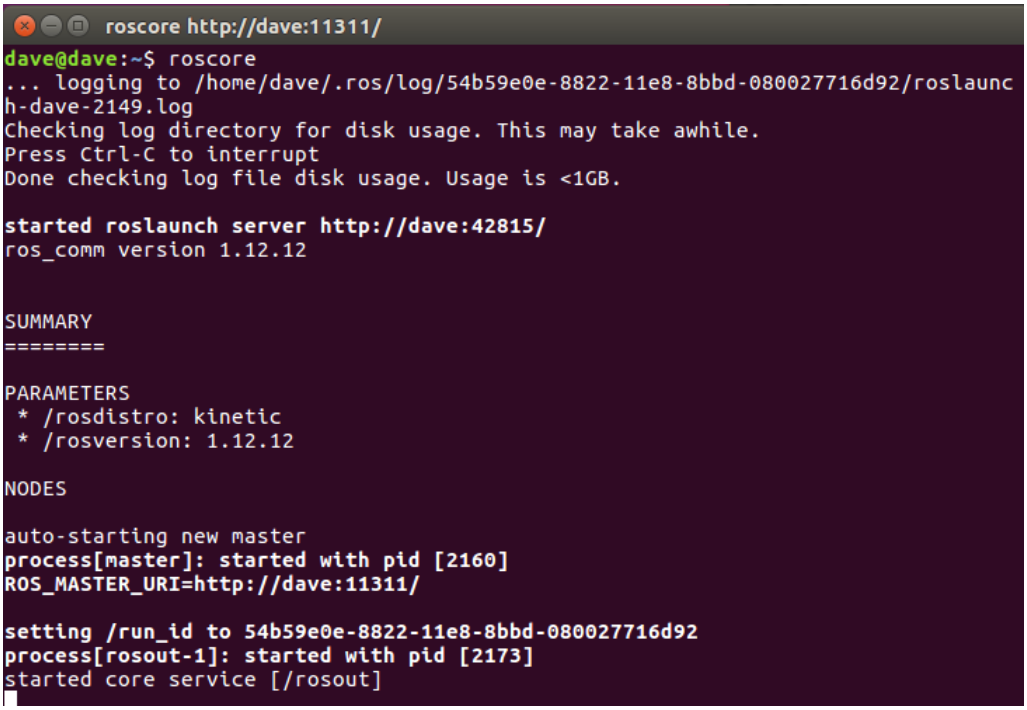
```
(...)

void agent_init()
{

      Py_Initialize();
      int argc = 1;
        char * argv[1];
      PySys_SetArgv(argc, argv);

      PyObject *rospy = PyImport_ImportModule("rospy");
      if (!rospy)
      {
          PyErr_Print();
          printf("ERROR in ROSPY\n");
          exit(1);
      }

      PyObject *init_node = PyObject_GetAttrString(rospy, "init_node");
      if (!init_node)
      {
          PyErr_Print();
          printf("ERROR in INIT_NODE\n");
          exit(1);
      }

      PyObject *init_node_args = PyTuple_New(1);
      PyObject *node_name = PyString_FromString("dron_camera");
      PyTuple_SetItem(init_node_args, 0, node_name);
      PyObject_CallObject(init_node, init_node_args);
      printf("Node created\n");

      PyObject * pName = PyString_FromString(AGENT_MODULE_NAME);

      pModule = PyImport_Import(pName);

      Py_DECREF(pName);

      if (pModule == NULL)
{
                PyErr_Print();
                error("Failed to load %s", AGENT_MODULE_NAME);
      }

      pFunc = PyObject_GetAttrString(pModule, AGENT_FUNCTION_NAME);

      if (!(pFunc && PyCallable_Check(pFunc)))
{
                if (PyErr_Occurred())
{
                          PyErr_Print();
                }
                error("Cannot find function %s", AGENT_FUNCTION_NAME);
      }

      pArgs = PyTuple_New(12);

      pResult = PyTuple_New(5);
}

(...)
```

- ***autopylot_agent.py***

```
(...)

import numpy as np
import cv2
import rospy
from sensor_msgs.msg import Image
import cv_bridge

pub = rospy.Publisher('frame', Image, queue_size=10)
bridge = cv_bridge.CvBridge()

def action(img_bytes, img_width, img_height, is_belly, \
    ctrl_state, vbat_flying_percentage, theta, phi, psi, altitude, vx, vy):

    # Report navigation data
    print('ctrl state=%6d battery=%2d%% theta=%+f phi=%+f psi=%+f altitude=%+3d vx=%f vy=%+f' % \
                          (ctrl_state, vbat_flying_percentage, theta, phi, psi, altitude, vx, vy))

    # Create full-color image from bytes
    image_cv = np.frombuffer(img_bytes, np.uint8)
    image_cv = np.ndarray.reshape(image_cv, (img_height, img_width, 3))
    image_ros = bridge.cv2_to_imgmsg(image_cv)
    pub.publish(image_ros)

    # Set up commands for a clockwise turn
    zap = 0
    phi = 0
    theta = 0
    gaz = 0
    yaw = 1

    return (zap, phi, theta, gaz, yaw)
```

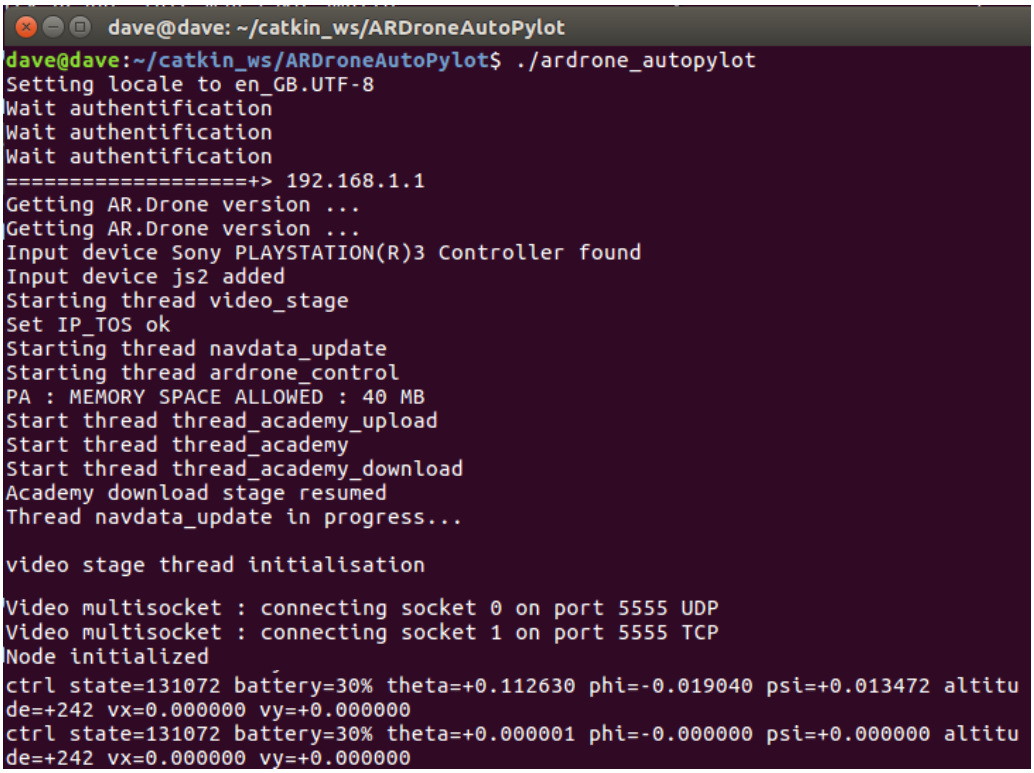
Una vez llevadas a cabo estas modificaciones procedemos a compilar mediante el comando *make*, obteniendo finalmente, el ejecutable que nos permitirá tanto el control del dron mediante el joystick como la obtención de la imagen del dron ya encapsulada en el tópico */frame*

El siguiente paso consiste en el lanzamiento del nodo Ros Master. Para ello sencillamente abriremos un terminal en el PC Virtual 1 y ejecutaremos el comando *roscore*. En la siguiente imagen podemos ver la información que nos muestra el sistema una vez hemos lanzado el ROS Master.



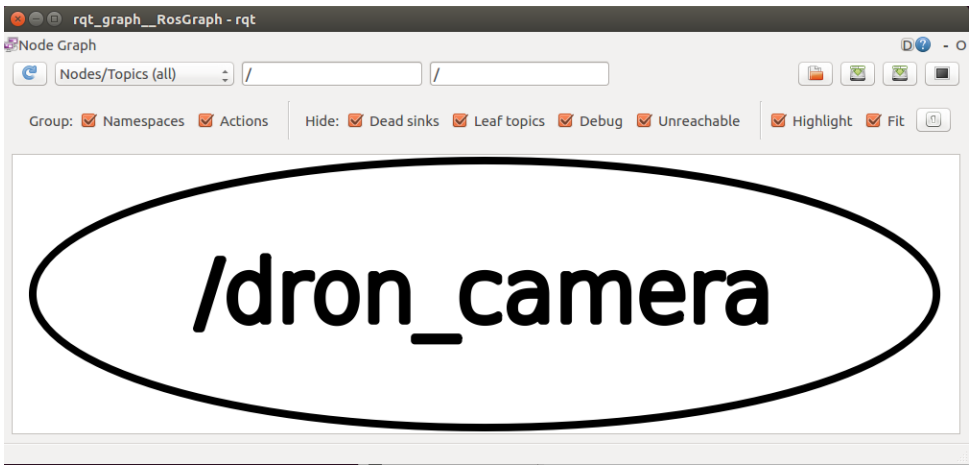
Inicialización del ROS Master

Una vez iniciado el ROS Master, procedemos a ejecutar el sistema ardrone_autopylot, que nos permite el pilotaje del dron con el joystick y leer algunas variables de navegación del dron y el porcentaje de batería restante.



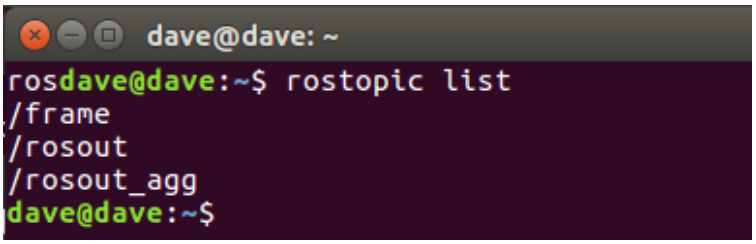
Ejecución del sistema de navegación *ardrone_autopylot*

Al lanzar el ejecutable, se lanza asimismo el nodo */dron_camera*, que publica el tópico */frame*. Gracias a la ejecución del comando *rosrun rqt_graph rqt_graph* podemos ver que efectivamente se crea el nodo */dron_camera.*



Nodo que publica las imágenes de la cámara del dron

Gracias al comando *rostopic list* podemos ver los tópicos activos en el sistema ROS. Podemos comprobar que el tópico /frame ha sido creado con éxito.



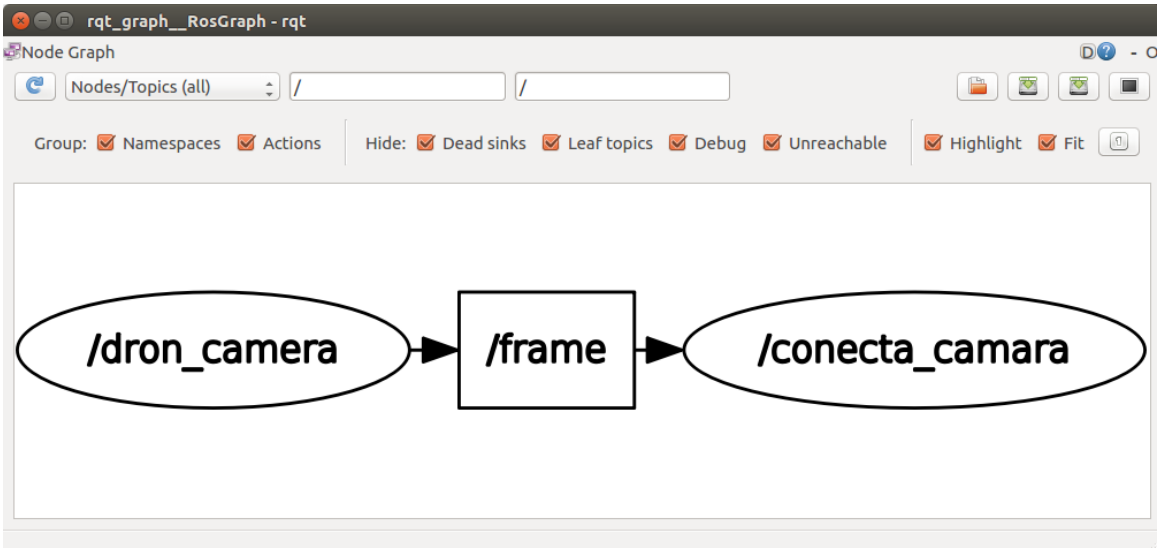
Lista de tópicos activos en el sistema ROS

Mediante el comando *rostopic in*fo podemos comprobar las características del tópico con el que trabajamos. En nuestro caso se trata de un mensaje de tipo imagen. Podemos leer cuáles son los nodos que publican el tópico y cuáles se suscriben al mismo. En la siguiente imagen podemos ver el caso particular después de lanzar el driver y al lanzar asimismo un script sencillo en Python que nos permite ver las imágenes de la cámara mediante subscripción al tópico mediante el nodo *conecta_camara*, como veremos en la imagen posterior gracias a *rqt_graph*.

```
dave@dave:~$ rostopic info /frame
Type: sensor_msgs/Image

Publishers:
 * /dron_camera (http://dave:46685/)

Subscribers:
 * /conecta_camara (http://dave:34541/)
```

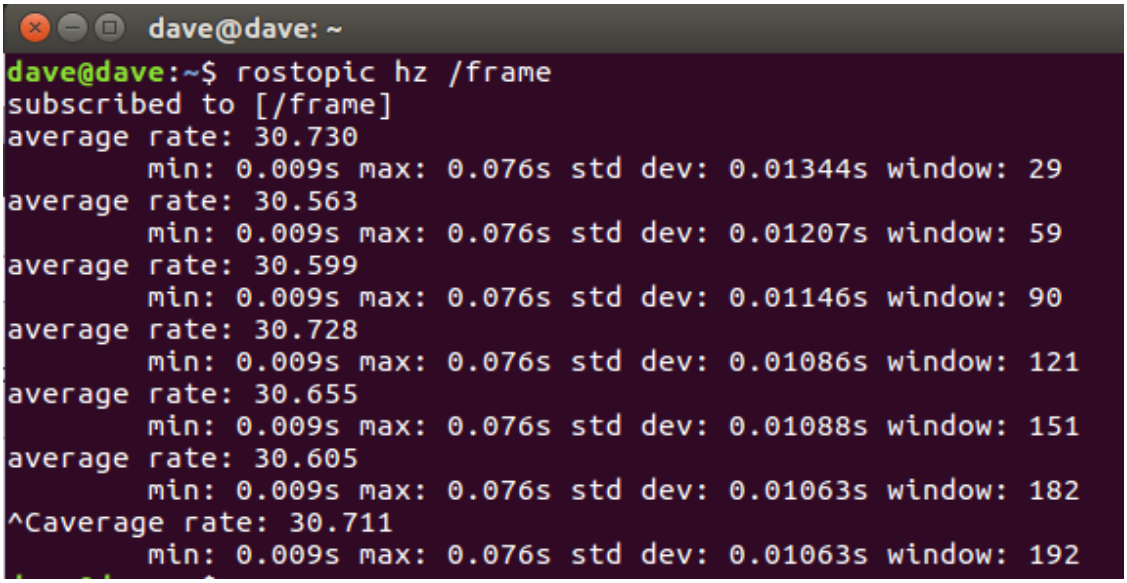
Propiedades del tópico */frame*

La siguiente figura nos muestra el esquema ROS del ejemplo citado en el párrafo anterior, el cual usamos de paso intermedio para comprobar que el tópico recibe imágenes correctamente de la cámara del dron antes de pasar a trabajar con redes neuronales.



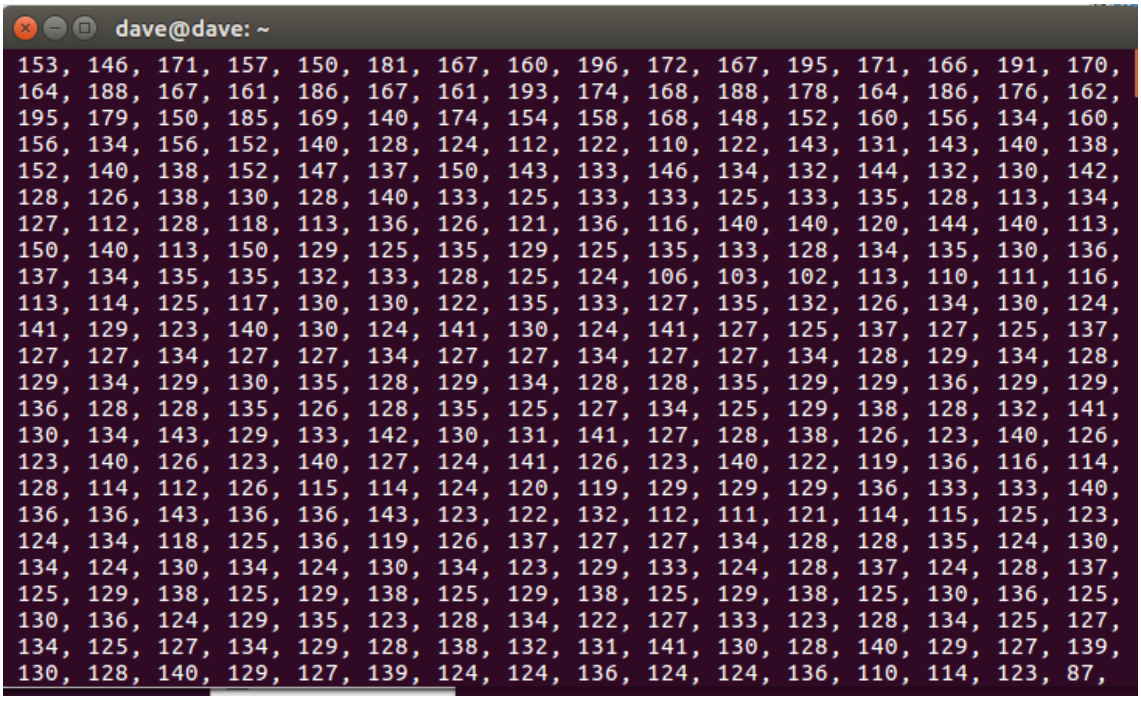
Esquema ROS de visualización de la cámara del dron

Otro comando ROS que se hace imprescindible para el desarrollo de este proyecto es *rostopic hz,* el cual nos permite conocer el número de fps del tópico que consultamos. En nuestro caso, consultando directamente sobre el tópico /frame sobre el PC Virtual , vemos que obtenemos en torno a 30fps.



Ratio de imágenes (fps) del */frame* en el PC Virtual 1

Un último comando que nos puede resultar interesante es *rostopic echo*, el cual nos permite comprobar la información transmitida por un tópico a tiempo real. La siguiente imagen muestra el contenido del tópico */frame.*



Datos procedentes de la lectura del tópico */frame*

ANEXO III. Preparación del entorno de la Jetson TX2.

Al introducir un nuevo hardware para el procesado de la red neuronal en la prueba 2, tenemos que preparar el entorno del mismo para conseguir hacer funcionar el sistema. El proceso inicialmente resulta tedioso, fundamentalmente debido a errores de versiones y compatibilidad. Se hace necesario, como se decía en apartados anteriores extremar la atención en todo el software y librerías que se instalen. Conviene aclarar también que el sistema operativo con que trabaja la Jetson TX2 no es convencional y por esta razón la instalación de software puede que tenga que hacerse de repositorios específicos. Es de vital importancia también extremar la atención a la hora de ejecutar comandos de actualización de Ubuntu dado que el Sistema Operativo (Tegra-Ubuntu) podría quedar inestable.

```
sudo apt-get update
sudo apt-get upgrade
```

Los pasos que seguimos para preparar el entorno en la Jetson TX2 son:

1. **Instalación de JetPack 3.2**

Una de las grandes ventajas que tiene hacer uso de un sistema embebido como Jetson TX2 es que la instalación básica del entorno de software del mismo está ya preparado en un paquete de software. En el caso de nuestro proyecto hacemos uso del JetPack 3.2, que nos instala el software base que se muestra a continuación.

JetPack 3.2, además del sistema operativo (Tegra-Ubuntu 16.04) y OpenCVv3.1.1, instala (*JETSON TX2, 2018)*):

TensorRT 4.0	TensorRT mejora el aprendizaje automático de los proyectos. Es de aplicación en clasificación de imágenes, segmentación y detección de objetos con redes neuronales. También acelera la inferencia del aprendizaje profundo y reduce el tiempo de funcionamiento de memoria en redes convolucionales.
cuDNN 7.1.5	La librería CUDA Deep Neural Network proporciona primitivas de alto rendimiento. Incluye soporte para convoluciones, activación de funciones y transformadas de tensores.
VisionWorks 1.6	VisionWorks es un paquete de software de desarrollo para visión por computador (CV) y procesado de imágenes. Incluye VPI (Vision Programming Interface), un conjunto de primitivas para uso de desarrolladores de CUDA.
CUDA 9.0	El kit de desarrollo CUDA Toolkit proporciona un exhaustivo entorno para el desarrollo de aplicaciones de aceleración en C and C++
Multimedia API	Jetson Multimedia API proporciona interfaces de programación de aplicaciones de bajo nivel para una aplicación flexible, por ejemplo para cámaras, sensores…
L4T	NVIDIA® Tegra® Linux Driver Package proporciona una plataforma de desarrollo en la Jetson.
Development Tools	NVIDIA System Profiler 4.0 es un muestreador CPU multi-core que monitoriza contantemente con el fin de mejorar el rendimiento general de la aplicación. . Tegra Graphics Debugger 2.5 es una consola que permite a los desarrolladores buscar errores y depurar OpenGL ES 2.0, 3.0, 3.1 y 3.2, OpenGL 4.3-4.6, para llegar a máximo rendimiento de la Jetson.

2. Instalación de ROS Kinetic

Para llevar a cabo la instalación de ROS Kinetic no lo haremos por el procedimiento habitual a través de la página oficial de ROS, sino a partir del repositorio *JETSONHACKS (2016-2018)*llevando a cabo la clonación del repositorio y después la compilación del mismo.

3. Instalar cv_bridge:

Se hace necesaria la instalación del paquete cv_bridge para poder convertir la imagen de formato ROS a formato OpenCV y viceversa. Para la instalación ejecutamos este comando:

```
sudo apt-get install ros-kinetic-cv-bridge
```

4. Instalar TensorFlow 1.7

Para ello seguimos los pasos que indica *Lee (2018)*

5. Instalar image_transport:

Image_transport es necesario para publicar y subscribir a imágenes a través de ROS. En [41] vemos con más detalle información acerca de su uso y de su instalación, que llevaremos a cabo con el siguiente comando:

```
sudo apt-get install ros-kinetic-compressed-image-transport
```

ANEXO IV. Re-entrenamiento del modelo en TensorFlow.

Este anexo recoge tanto los scripts como los comandos utilizados para re-entrenar la red neuronal en TensorFlow.

A.IV.1. Creación de TFRecord.

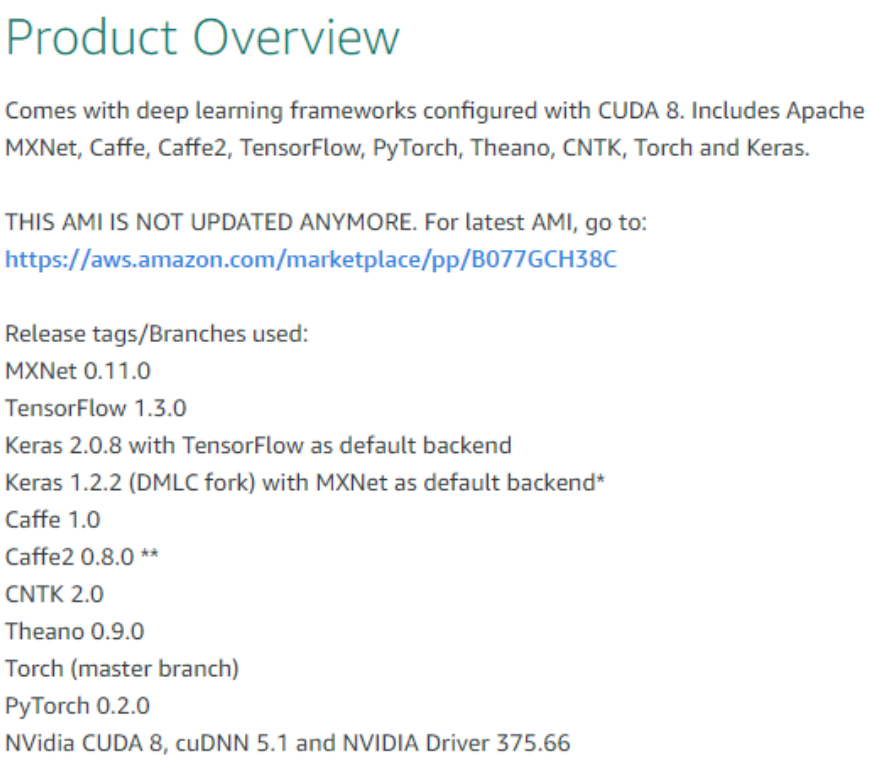
El siguiente script nos permite crear el archivo *tf.record*, que usaremos posteriormente para el re-entrenamiento. Este script lleva a cabo el proceso de encontrar aquellas imágenes que contienen las etiquetas *stop* o *yield* y las conjuga con sus etiquetas.

```
from __future__ import division
import io
import pandas as pd
import TensorFlow as tf
from PIL import Image

import sys
sys.path.append('./models/research')
from object_detection.utils import dataset_util


def class_text_to_int(label):
    if label == 'stop':
        return 1
    elif label == 'yield':
        return 2
    else:
        return 0

# Creamos el tf.record para ambas clases {stop y yield}
writer = tf.python_io.TFRecordWriter('tf_record/stop_yield.record')

for video in range(11):

    if video == 7: continue

    tags_df = pd.read_csv('vid{}/frameAnnotations.csv'.format(video), sep=';')

#Busca en las anotaciones bien ‘stop’ o bien ‘yield’
    for index, row in tags_df.iterrows():
        if row['Annotation tag'] == 'stop' or 'yield':

            filename = row['Filename']

            with tf.gfile.GFile('vid{}/{}'.format(video, filename), 'rb') as fid:
                encoded_jpg = fid.read()
            encoded_jpg_io = io.BytesIO(encoded_jpg)
            image = Image.open(encoded_jpg_io)
            width, height = image.size
            image_format = b'png'
```

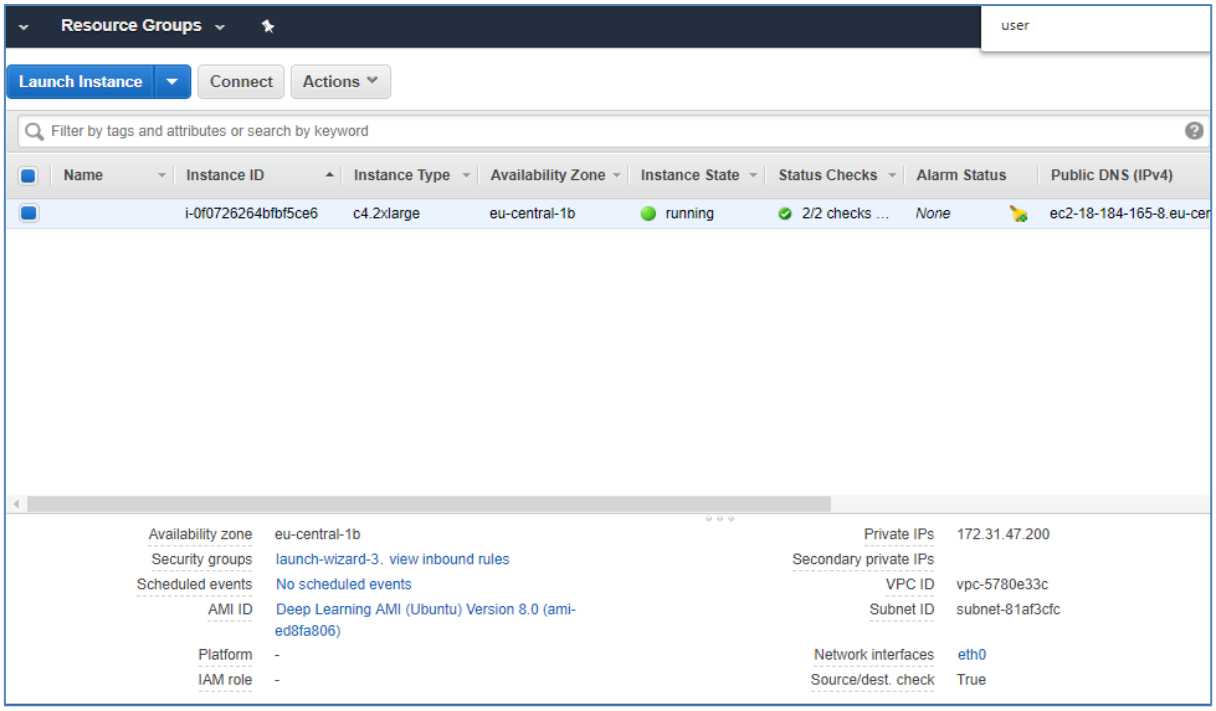
```
# Lo hace por porcentaje

            xmins = [row['Upper left corner X']/width]
            xmaxs = [row['Lower right corner X']/width]
            ymins = [row['Upper left corner Y']/height]
            ymaxs = [row['Lower right corner Y']/height]

            classes_text = [row['Annotation tag']]
            classes = [class_text_to_int(row['Annotation tag'])]

            tf_example = tf.train.Example(features=tf.train.Features(feature={
                'image/height': dataset_util.int64_feature(height),
                'image/width': dataset_util.int64_feature(width),
                'image/filename': dataset_util.bytes_feature(filename),
                'image/source_id': dataset_util.bytes_feature(filename),
                'image/encoded': dataset_util.bytes_feature(encoded_jpg),
                'image/format': dataset_util.bytes_feature(image_format),
                'image/object/bbox/xmin': dataset_util.float_list_feature(xmins),
                'image/object/bbox/xmax': dataset_util.float_list_feature(xmaxs),
                'image/object/bbox/ymin': dataset_util.float_list_feature(ymins),
                'image/object/bbox/ymax': dataset_util.float_list_feature(ymaxs),
                'image/object/class/text': dataset_util.bytes_list_feature(classes_text),
                'image/object/class/label': dataset_util.int64_list_feature(classes),
            }))

            writer.write(tf_example.SerializeToString())

writer.close()
```

A.IV.2. Reentrenamiento de la red neuronal con TensorFlow.

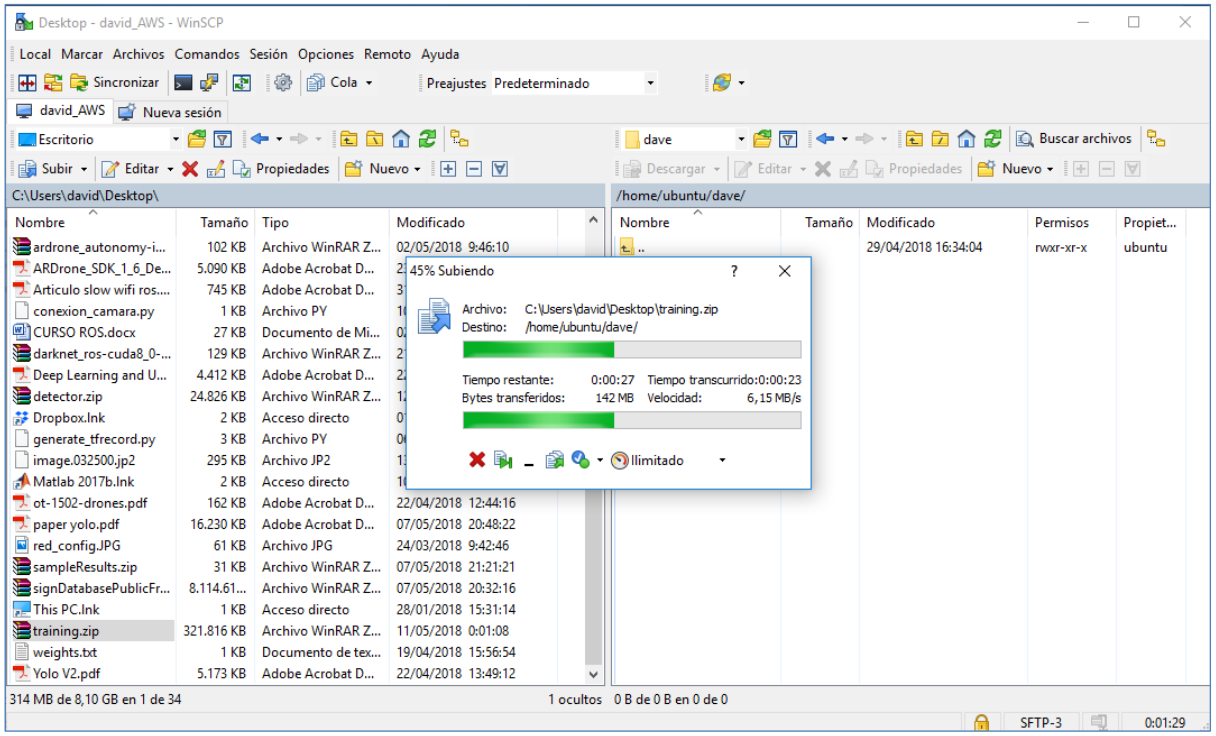
El re-entrenamiento de la red neuronal con TensorFlow se lleva a cabo haciendo uso del servicio EC2, de AWS de Amazon. Tras crear una cuenta lanzamos una instancia con estas características:



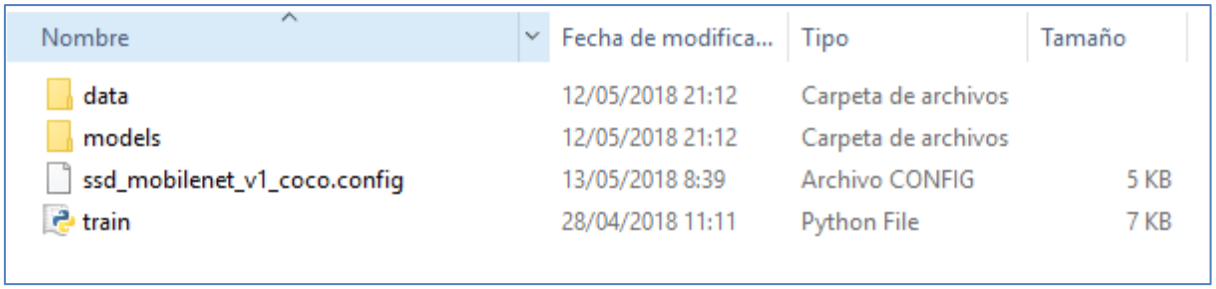
A continuación iniciamos la instancia mediante la consola, obteniendo la IP pública, que utilizaremos para conectarnos en remoto al sistema.



Nos conectamos a la máquina remota gracias a la aplicación WinSCP, que nos va a permitir visualizar el contenido de la máquina remota en formato árbol. Subiremos nuestra estructura de carpetas lista para entrenar.

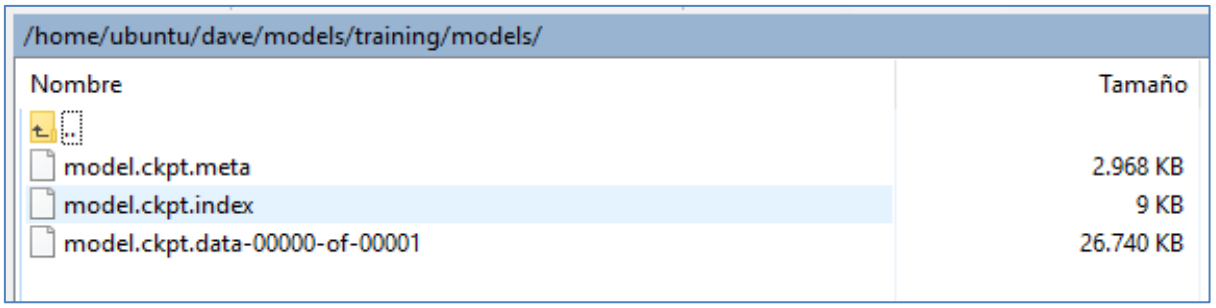


La estructura de carpetas que subimos tiene este aspecto:



Nombre	Fecha de modifica...	Tipo	Tamaño
data	12/05/2018 21:12	Carpeta de archivos	
models	12/05/2018 21:12	Carpeta de archivos	
ssd_mobilenet_v1_coco.config	13/05/2018 8:39	Archivo CONFIG	5 KB
train	28/04/2018 11:11	Python File	7 KB

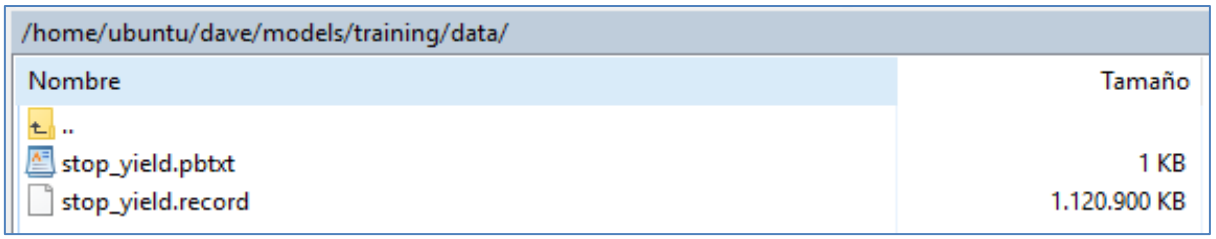
La carpeta *models* contendrá los archivos .ckpt originales del modelo ssd_mobilenet_v1_coco.



/home/ubuntu/dave/models/training/models/

Nombre	Tamaño
..	
model.ckpt.meta	2.968 KB
model.ckpt.index	9 KB
model.ckpt.data-00000-of-00001	26.740 KB

La carpeta *data* contendrá por un lado el archivo tf.record creado con anterioridad llamado stop_yield.record y por otro lado el archivo pbtxt que indica la id y el nombre de cada una de las clases a entrenar.



/home/ubuntu/dave/models/training/data/

Nombre	Tamaño
..	
stop_yield.pbtxt	1 KB
stop_yield.record	1.120.900 KB

El archivo stop_yield.pbtxt tiene una estructura como la que se muestra a continuación:

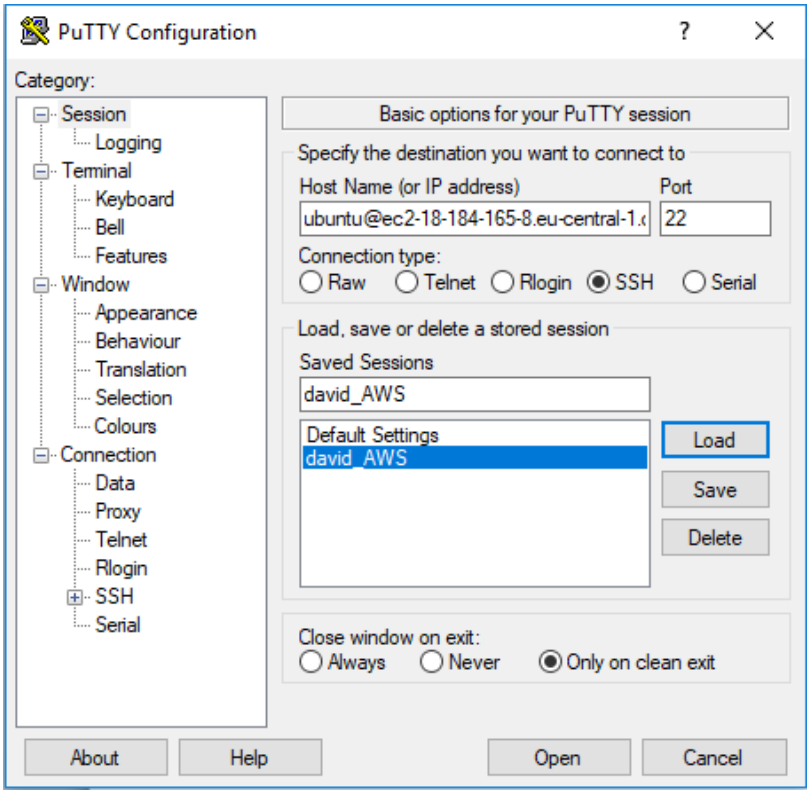
```
item {
    id: 1
    name: 'stop'
}
item {
    id: 2
    name: 'yield'
}
```

En lo que respecta al archivo sd_mobilenet_v1_coco.config, modificamos las siguientes variables:

- ✓ Número de clases: 2
- ✓ Número de pasos en el reentrenamiento: 6000

Asimismo indicamos la ubicación tanto de los archivos de entrenamiento como de los de evaluación. En el caso del trabajo que hemos desarrollado, hemos utilizado los mismos datos

Una vez definida la estructura y haciendo uso de la aplicación Putty, nos conectamos remotamente al sistema que tenemos contratado vía SSH.



Una vez conectados al sistema, habilitamos la máquina con TensorFlow preinstalado con el comando:

source activate TensorFlow_p27

Antes de proceder al entrenamiento, es importante que ejecutemos los siguientes comandos desde el directorio TensorFlow/models/research.

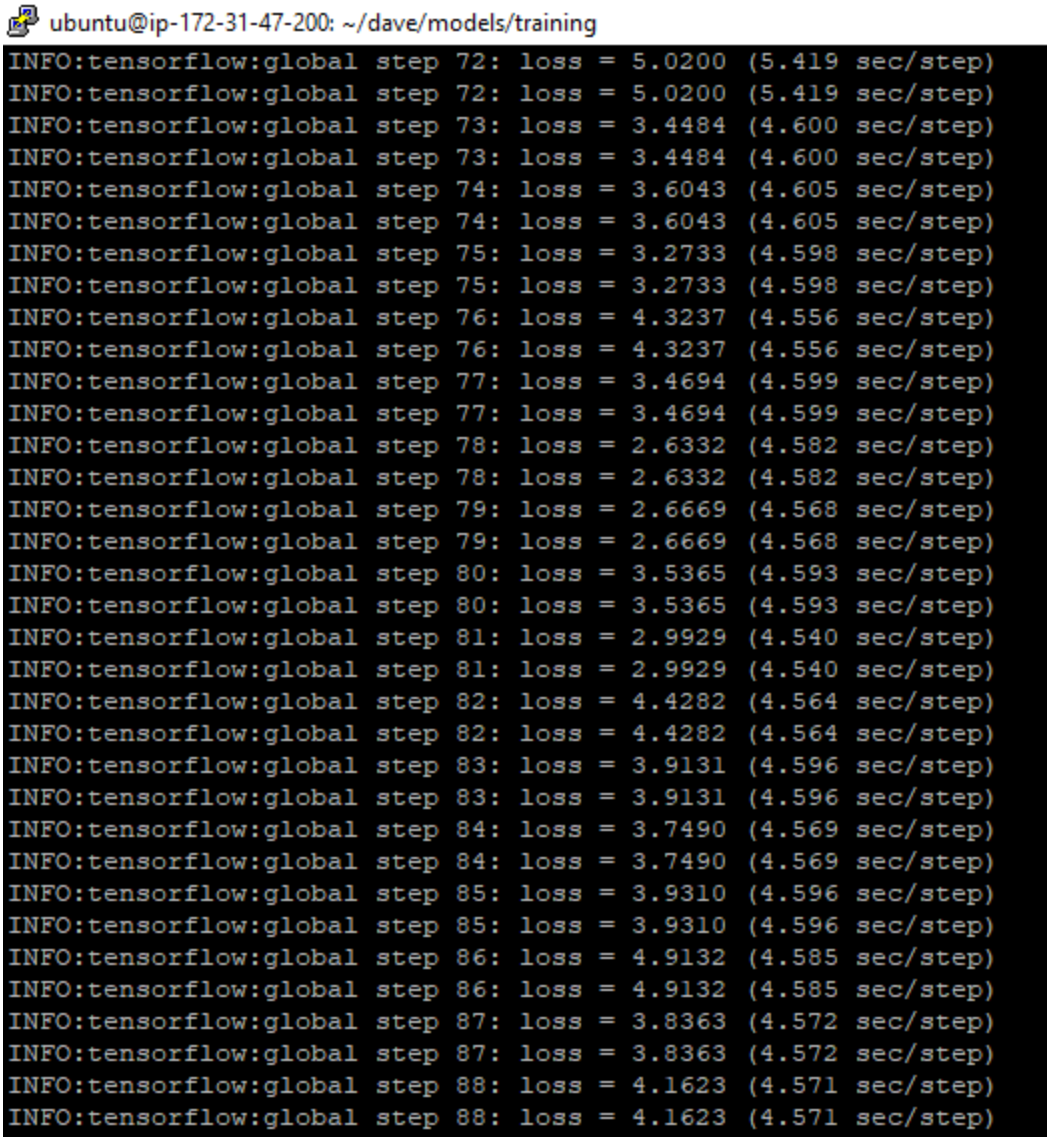
```
# From TensorFlow/models/research/
protoc object_detection/protos/*.proto --python_out=.
```

```
# From TensorFlow/models/research/
export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim
```

Una vez ejecutados estos comandos regresamos al directorio en que se encuentra el script train.py y ejecutaremos el siguiente comando:

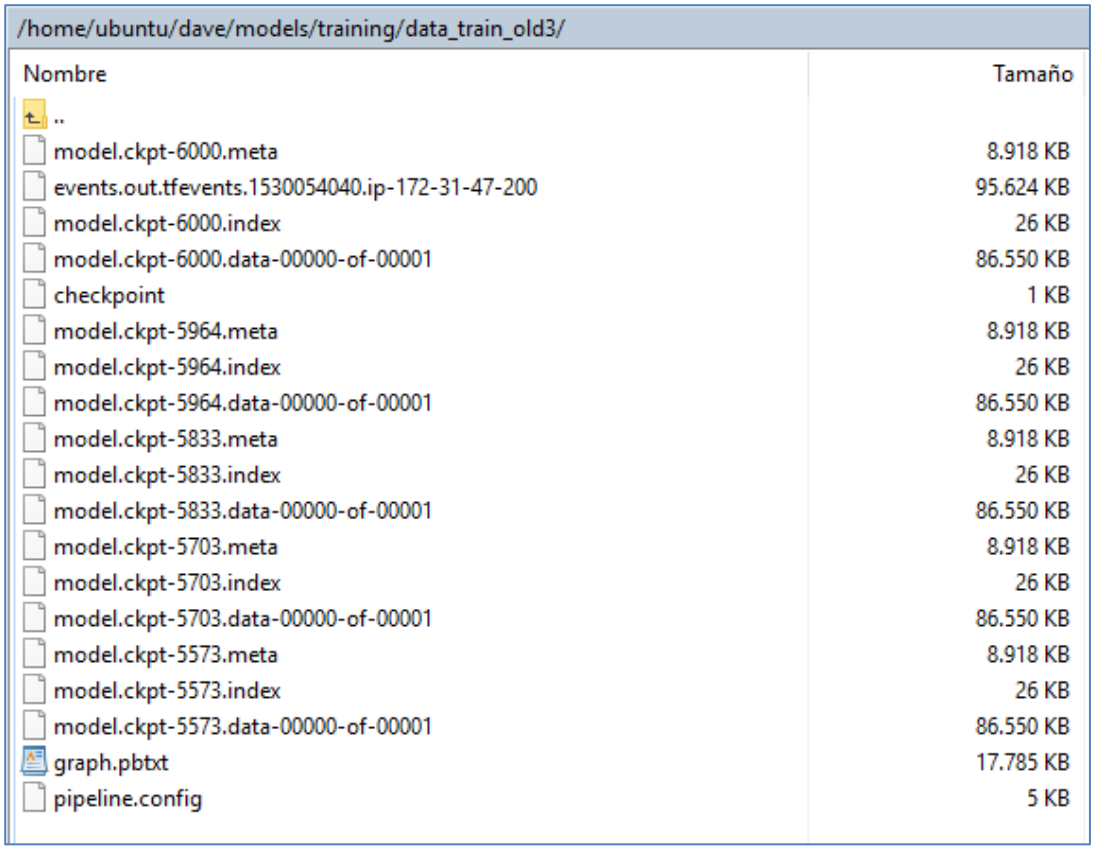
python train.py --logtostderr --train_dir=./data_train --
pipeline_config_path=ssd_mobilenet_v1_coco.config

De esta forma arrancamos el entrenamiento, la siguiente figura muestra los primeros pasos de unos de los entrenamientos que llevamos a cabo.



Una vez acabado el número de pasos indicados, en este caso 6000, obtenemos en el directorio *data_train:*

- ✓ Los archivos CheckPoint del proceso de entrenamiento (ckpt).
- ✓ Un archivo llamado pipeline.config
- ✓ Un archivo llamado graph.pbtxt
- ✓ Un archivo de salida de eventos.



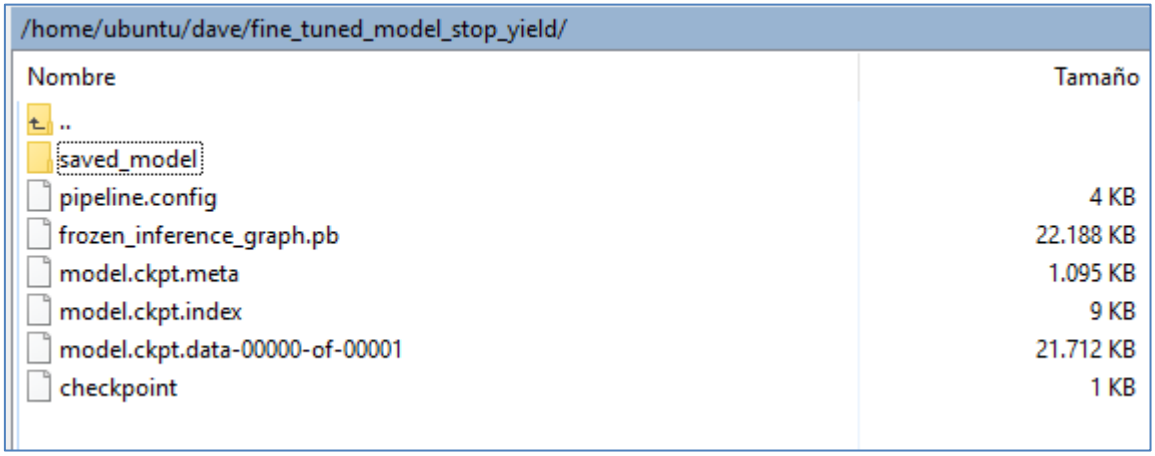
El último paso que queda por llevar a cabo es convertir los modelos Checkpoints (.ckpt) a un archivo .pb. Para ello haremos uso del script de TensorFlow *export_interference_graph.py*, el cual ubicaremos en el mismo directorio en que ejecutábamos el script train.py

A continuación ejecutamos el script.

python export_inference_graph.py --input_type image_tensor --pipeline_config_path /home/ubuntu/dave/models/training/data_train/pipeline.config --trained_checkpoint_prefix /home/ubuntu/dave/models/training/data_train/model.ckpt-6000 --output_directory /home/ubuntu/dave/fine_tuned_model_stop_yield

El resultado se ubicará en el archivo de salida, llamado *fine_tuned_model_stop_yield*, que será el que usaremos para probar en nuestro sistema, ejecutándolo en la GPU.

Finalmente, este es el resultado que obtenemos la carpeta de salida que indicábamos anteriormente.



De estos archivos, usaremos el llamado *frozen_interference_graph.pb*, con un tamaño aproximado de 22 MB para ejecutar en nuestro sistema de detección basado en TensorFlow y ssd_mobilenet_v1_coco reentrenado.

A.IV.3. Resultados del re-entrenamiento

Este apartado del anexo IV recoge el resultado del re-entrenamiento llevado a cabo con TensorFlow y el modelo SSD Mobilenet.

Una vez finalizado el re-entrenamiento y como se ve en el apartado anterior, se generan una serie de archivos:

- ✓ Los archivos CheckPoint del proceso de entrenamiento (ckpt).
- ✓ Un archivo llamado pipeline.config
- ✓ Un archivo llamado graph.pbtxt
- ✓ Un archivo de salida de eventos.

Se va a hacer uso de Tensorboard, un visualizador web que nos permite analizar en profundidad el resultado del reentrenamiento.

Para poder acceder a Tensorboard es necesario ubicarse en la carpeta anterior a la que se ubican los archivos indicados anteriormente, en nuestro caso:

/home/ubuntu/dave/models/training/

Posteriormente ejecutamos TensorBoard mediante el comando:

tensorboard –logdir [directorio_checkpoints]

```
nvidia@tegra-ubuntu:~/Desktop/Redacción TFM_stop_works/Training_old$ tensorboard
 --logdir data_train
2018-08-06 17:13:24.719499: I tensorflow/stream_executor/cuda/cuda_gpu_executor.
cc:865] ARM64 does not support NUMA - returning NUMA node zero
2018-08-06 17:13:24.719679: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1
344] Found device 0 with properties:
name: NVIDIA Tegra X2 major: 6 minor: 2 memoryClockRate(GHz): 1.3005
pciBusID: 0000:00:00.0
totalMemory: 7.66GiB freeMemory: 6.23GiB
2018-08-06 17:13:24.719730: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1
423] Adding visible gpu devices: 0
2018-08-06 17:13:27.224312: I tensorflow/core/common_runtime/gpu/gpu_device.cc:9
11] Device interconnect StreamExecutor with strength 1 edge matrix:
2018-08-06 17:13:27.224384: I tensorflow/core/common_runtime/gpu/gpu_device.cc:9
17]      0
2018-08-06 17:13:27.224413: I tensorflow/core/common_runtime/gpu/gpu_device.cc:9
30] 0:   N
TensorBoard 1.7.0 at http://tegra-ubuntu:6006 (Press CTRL+C to quit)
```

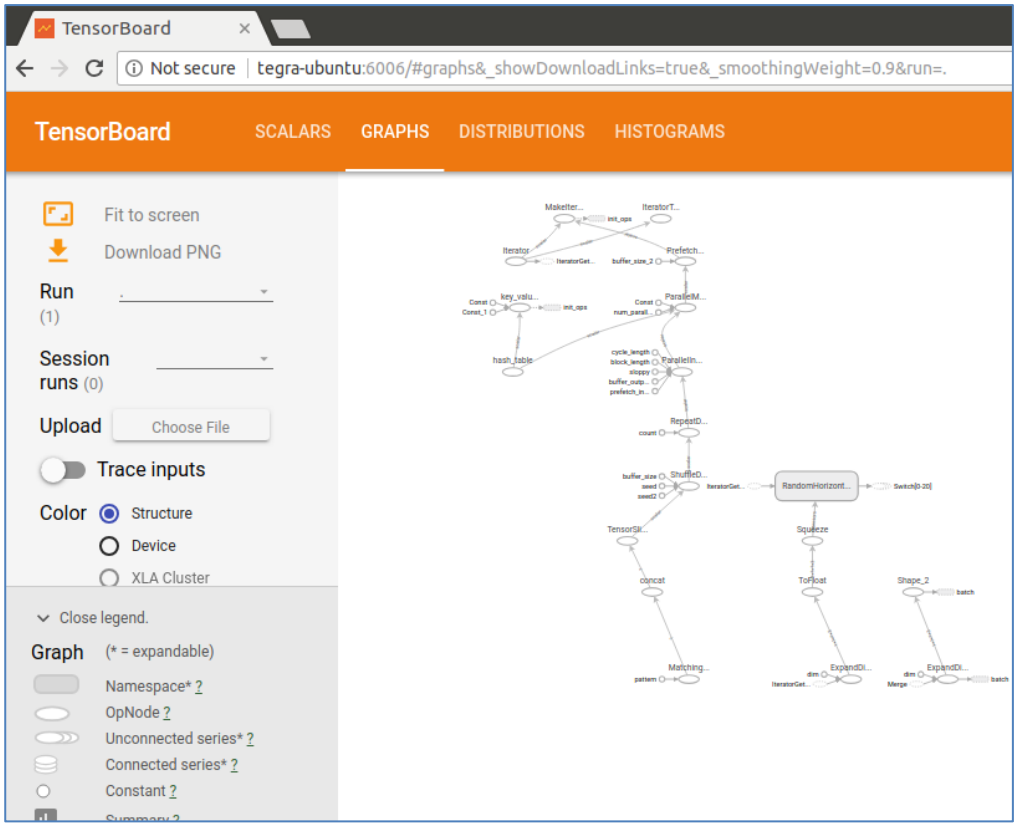
Inicialización de TensorBoard

Finalmente, para acceder a Tensorboard, basta con pulsar CTRL + el enlace *http://tegra-ubuntu:6006*

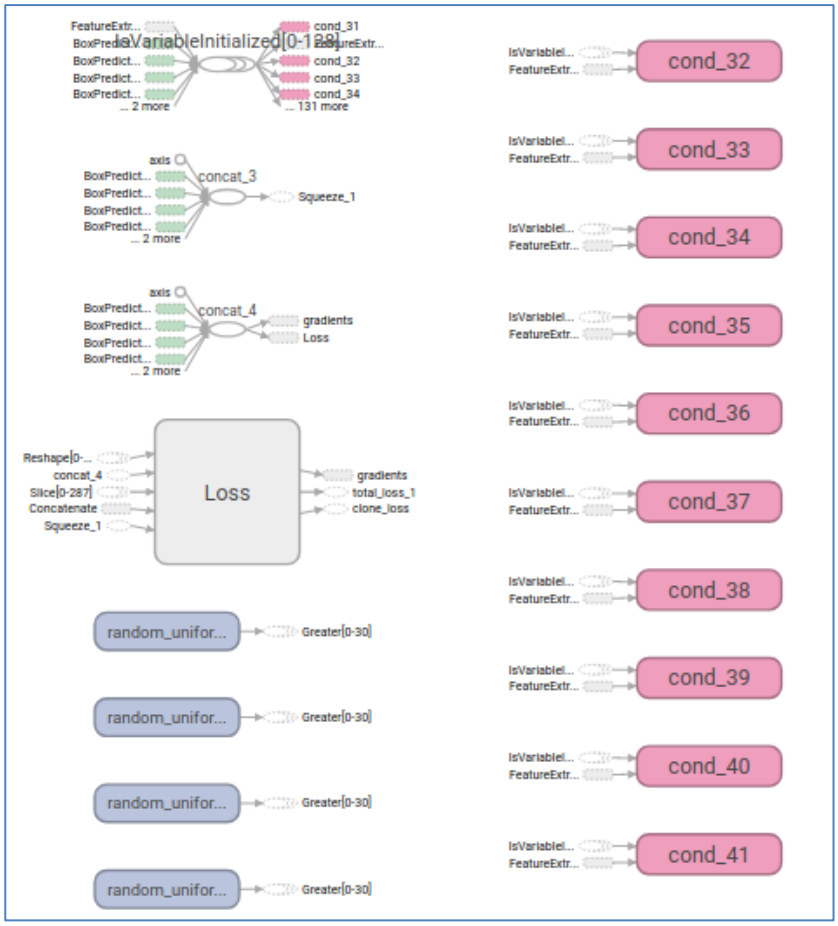
Al abrir Tensorboard, vemos que hay cuatro menús principales:

- Scalars
- Graphs
- Distributions
- Histograms

Inicialmente nos centramos en el menú Graphs, en el que podemos ver el grafo que se ha construido con la red neuronal. En las siguientes figuras podemos ver detalles tanto del grafo principal como de una parte de los auxiliares, donde se observar la parte del grafo destinado al cálculo de la función de coste.



Detalle del Grafo en TensorFlow

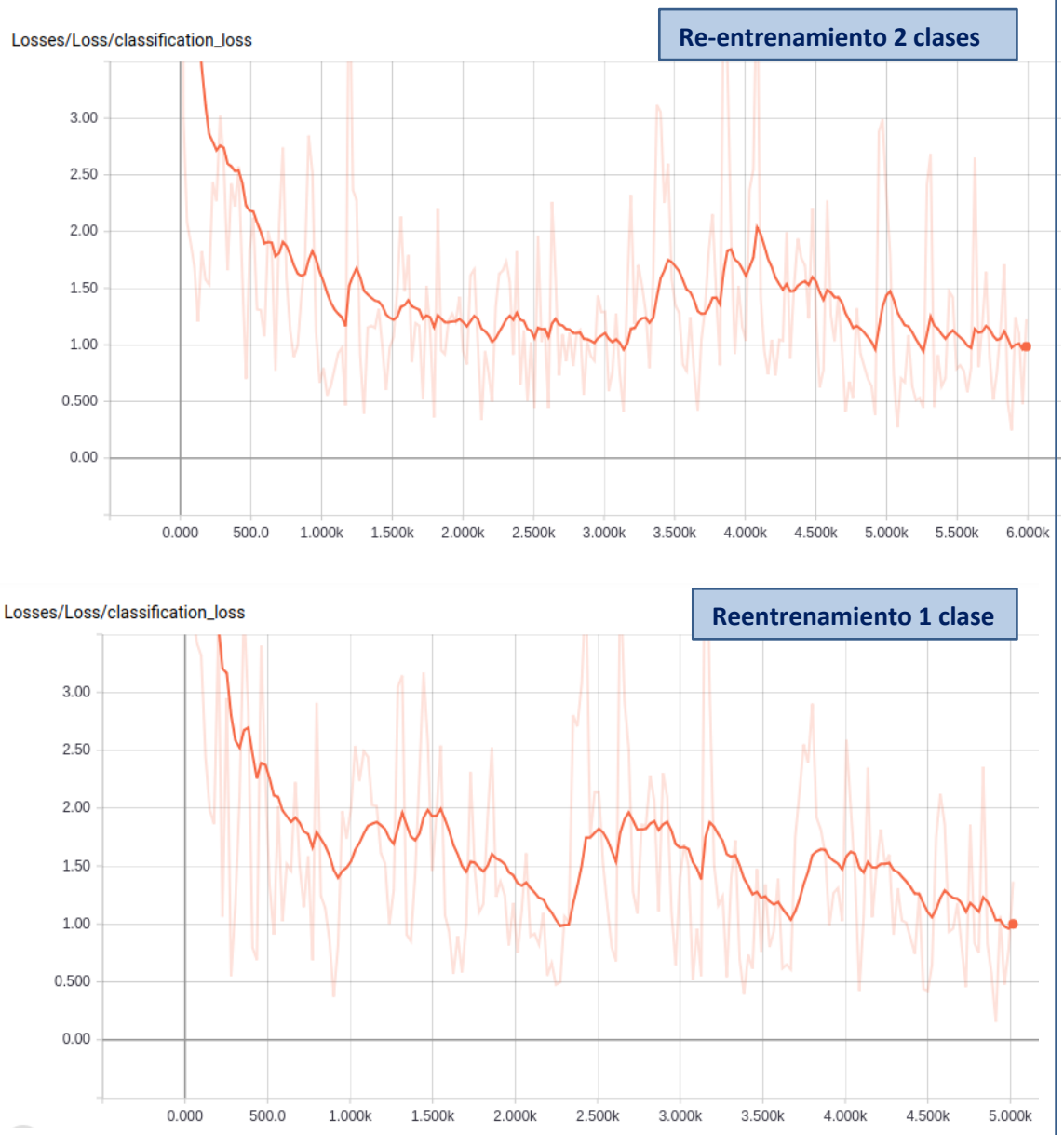


Detalle Grafo en TensorFlow destinado al cálculo de la función de coste

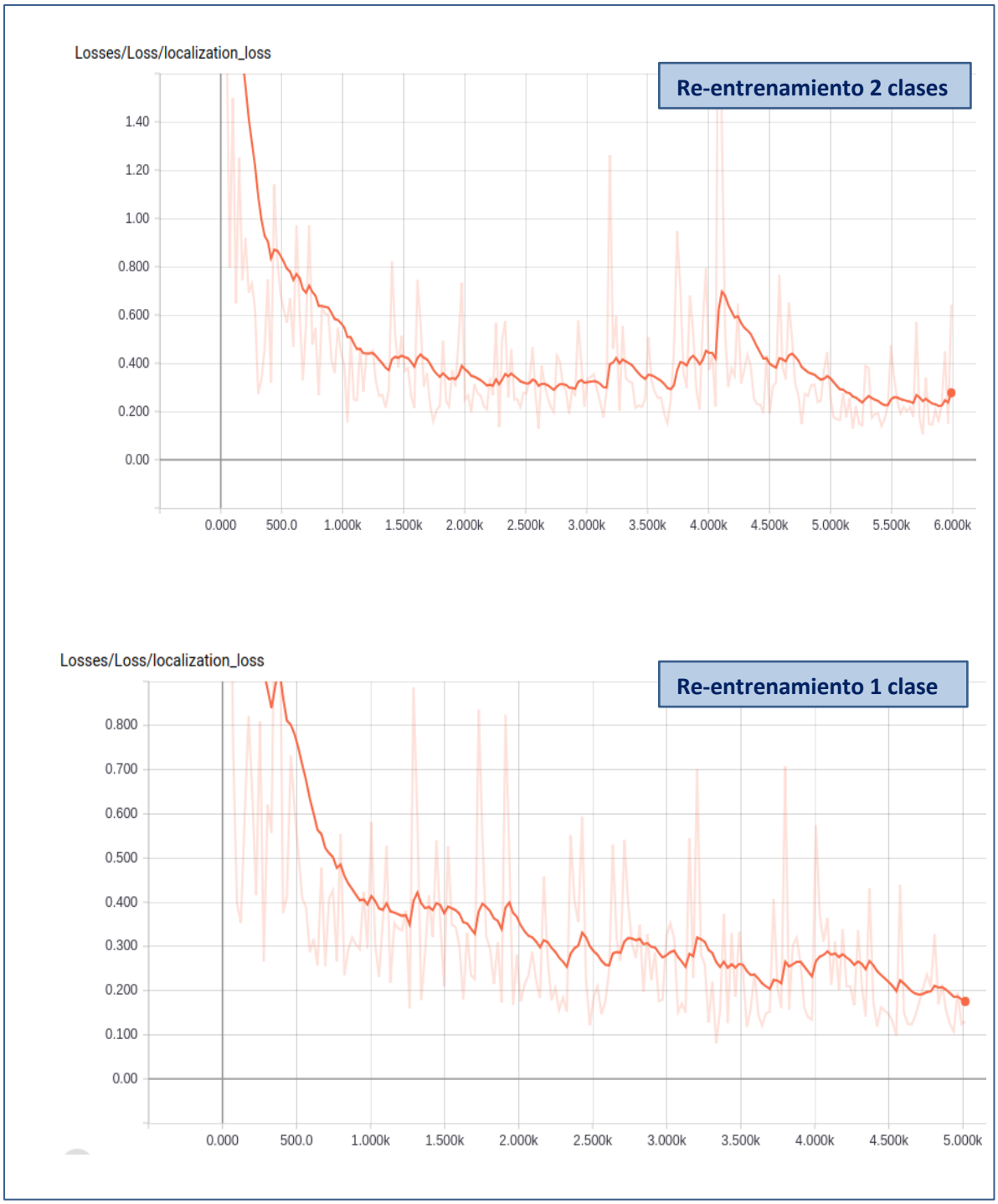
Dada la complejidad que presenta el grafo, no se profundiza más sobre el mismo, nos centraremos únicamente en el apartado *Scalars* y dentro de este, en la opción *Losses*, que nos permitirá visualizar los gráficos obtenidos a partir de la función de coste, generados durante el re-entrenamiento. Compararemos los gráficos de los resultados de dos intentos de re-entrenamientos llevados a cabo:

- Reentrenamiento basado en imágenes de señales de stop y de ceda el paso.
- Reentrenamiento basado en imágenes de señales de stop.

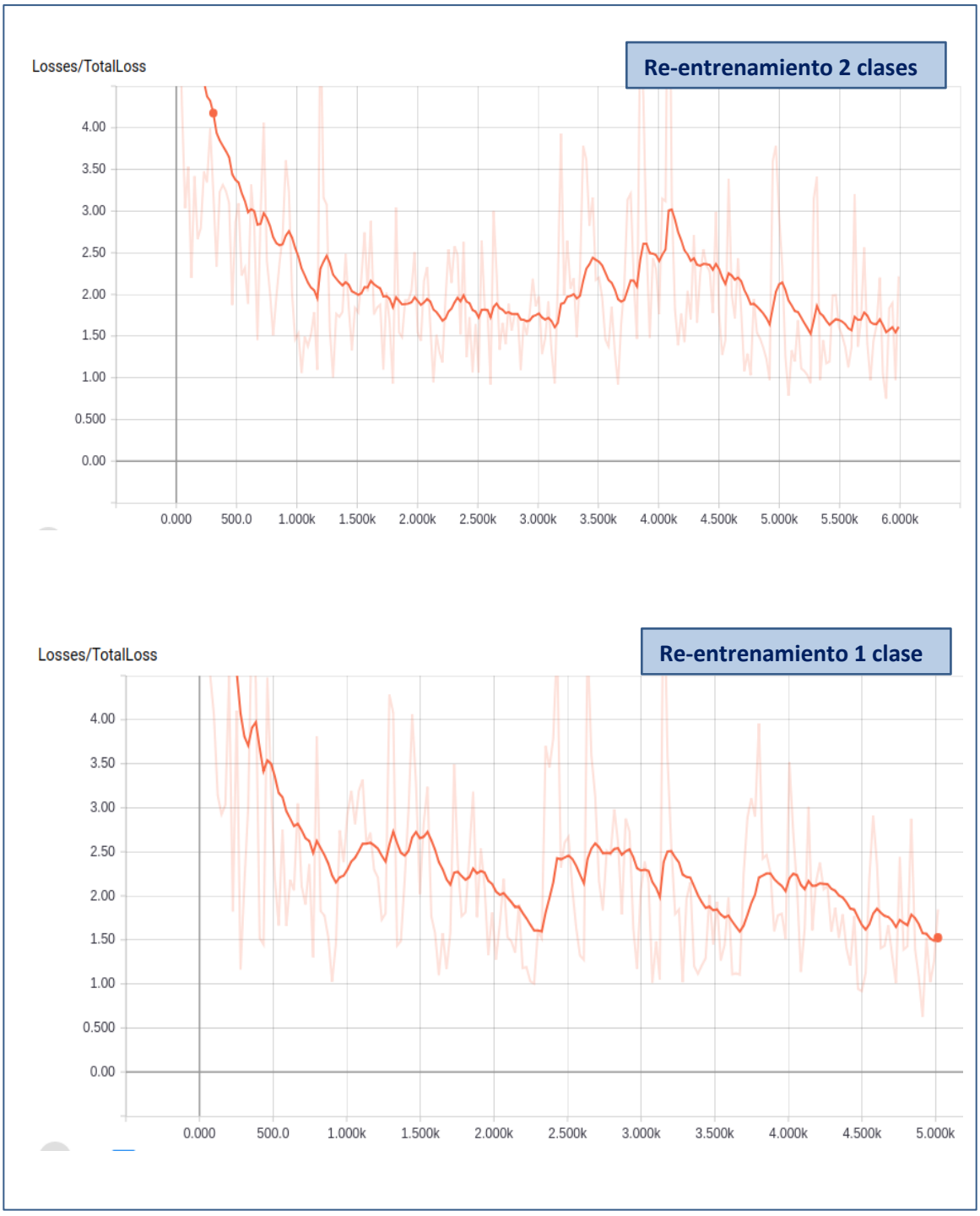
Como se mencionaba en el apartado 3.2.2.2, pese a que el re-entrenamiento con dos clases finaliza correctamente, no se consigue detectar ninguna de las dos clases al utilizar el modelo reentrenado, trataremos de explicarlo con las figuras que se muestran a continuación.



Gráficos de la función de coste en la clasificación



Gráficos de la función de coste en la localización



Gráficos de la función de coste total

Todos los gráficos mostrados anteriormente se suavizan con un factor de 0.9 para poder visualizar mejor el trazado de la función. Se trata de un filtro lineal que se aplica de la siguiente forma, de acuerdo a [34]:

```
def smooth(scalars, weight):  # Weight between 0 and 1
    last = scalars[0]  # First value in the plot (first timestep)
    smoothed = list()
    for point in scalars:
        smoothed_val = last * weight + (1 - weight) * point  # Calculate smoothed value
        smoothed.append(smoothed_val)                        # Save it
        last = smoothed_val                                  # Anchor the last smoothed value

    return smoothed
```

Como se indica en el apartado 2.2.1, la función de coste del reentrenamiento responde a una suma ponderada de los costes de localización (loc) y de los costes de confianza en la detección (conf).

$$L(x, c, l, g) = \frac{1}{N}\left(L_{conf}(x, c) + \alpha L_{loc}(x, l, g)\right)$$

En los gráficos de la función de coste se puede apreciar que el trazado de la misma corresponde a una combinación de las funciones de la función de coste en la clasificación y en la localización.

Si observamos los gráficos correspondientes a la clasificación, vemos que el reentrenamiento con dos clases presenta valores más bajos que el reentrenamiento con una sola clase en menor número de iteraciones, aproximándose a 1 a aproximadamente en 1200 iteraciones. Sin embargo, el r-entrenamiento con una sola clase parece presentar un decrecimiento más regular hasta un mínimo relativo entre las 2000 y las 2500 iteraciones.

En lo que respecta a los gráficos de localización, observamos cómo el re-entrenamiento con 1 sola clase presenta un decrecimiento más homogéneo. El re-entrenamiento con 2 clases presenta crecimiento que se inicia a partir de las 3600 iteraciones, presentando un máximo relativo pasadas las 4000, a partir del cual comienza de nuevo a decrecer, aproximándose a valores próximos a 0.2.

Si nos fijamos en los gráficos de la función de coste total, vemos cómo el gráfico correspondiente al re-entrenamiento de dos clases (stop y ceda el paso) presenta un

máximo relativo una vez superadas las 4000 iteraciones, llegando a un valor de 3 aproximadamente, valor que ya tomaba una vez superadas las 500 iteraciones. El gráfico correspondiente al re-entrenamiento de una sola clase, pese a presentar algún pico, no parece presentar una crecimiento tan abrupto.

En ambos casos, probablemente se habría podido parar el re-entrenamiento entre las 2000 y las 2500 iteraciones, ya que parece que en este intervalo ha llegado a un mínimo con un decrecimiento relativamente homogéneo.

Si observamos el trazado de la función sin suavizar, vemos como en el caso de la función de coste total para el reentrenamiento con 2 clases parece presentar un fenómeno de sobreajuste (*overfitting*), tomando otro camino que converge hacia un punto engañoso.

El comportamiento de la función de coste en el caso del re-entrenamiento con dos clases puede deberse a que hemos utilizado muy pocas imágenes de ceda el paso (293) frente a las 1821 imágenes de stop. Esto sumado a que hemos hecho el reentrenamiento sin imágenes de validación, ha podido ser la causa por la que el modelo reentrenado para dos clases no detecta ninguna de las dos clases detectadas en las pruebas realizadas

ANEXO V. Script detector SSD Mobilenet.

```
import rospy
from sensor_msgs.msg import Image
import cv2, cv_bridge
import numpy as np
import copy
import tensorflow as tf
import time
import os

#Usamos la GPU con ID=0 y reservamos un 60 % de memoria de forma dinámica

os.environ["CUDA_VISIBLE_DEVICES"] = '0' #uUsar GPU con ID=0
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.6 # Reserva máxima del 60% de memoria MEM
config.gpu_options.allow_growth = True #Reserva de memoria dinámica

#Iniciamos el nodo ROS
rospy.init_node('detector')

# Habilitamos cv_bridge
bridge = cv_bridge.CvBridge()


# Cargamos el grafo congelado y construimos el grafo en TensorFlow a partir de los distintos tensores

from tensorflow.core.framework import graph_pb2
graph_def = graph_pb2.GraphDef()

with open("frozen_inference_graph.pb", "rb") as f:
    graph_def.ParseFromString(f.read())

tf.import_graph_def(graph_def, name='')

image_tensor = tf.get_default_graph().get_tensor_by_name('image_tensor:0') # Tensor de entrada
boxes_tensor = tf.get_default_graph().get_tensor_by_name('detection_boxes:0') # Coordenadas (cajas de
detección)
classes_tensor = tf.get_default_graph().get_tensor_by_name('detection_classes:0') # Clases (persona,
perro..)
scores_tensor = tf.get_default_graph().get_tensor_by_name('detection_scores:0') # Puntuaciones en la
detección.

# Abrimos el archivo que contiene las etiquetas

with open('labels.txt') as f:
    labels = f.read().splitlines()
labels = np.array(labels)

# Dibujamos las cajas de detección

def draw_boxes(img, box, label):
    ymin, xmin, ymax, xmax = box

    xmin = int(xmin * img.shape[1])
    xmax = int(xmax * img.shape[1])
    ymin = int(ymin * img.shape[0])
    ymax = int(ymax * img.shape[0])

    cv2.rectangle(img, (xmin, ymin), (xmax, ymax), (0, 0, 255), 3)

    font = cv2.FONT_HERSHEY_SIMPLEX
    cv2.putText(img, label, (xmin, ymax), font, 1.5, (0, 0, 255), 3)

    return img
```

```
# Cálculo de la inferencia. Presentamos las detecciones con puntuaciones superiores a 0,7

def inference(img):
    img_expanded = np.expand_dims(img, axis=0)


    (boxes, classes, scores) = sess.run([boxes_tensor, classes_tensor, scores_tensor], feed_dict={
      image_tensor: img_expanded
    })

    boxes, classes, scores = boxes[0], classes[0], scores[0]
    img_box = np.copy(img)
    for i, score in enumerate(scores):
        if score > 0.7:
            label = labels[int(classes[i])-1]
            box = boxes[i]

          t=time.time()

          print('Clase detectada:', label, 'Precision: {:.2%}'.format(score)) # Mostramos por
pantalla la clase detectada y la confianza

            img_box = draw_boxes(img_box, box, label) # Dibujamos las cajas y ponemos etiquetas

    return img_box

# Función principal. Mostramos la imagen y monitorizamos el tiempo de inferencia

def callback(frame_ros):
    frame_cv = bridge.imgmsg_to_cv2(frame_ros, "bgr8")
        t = time.time()
    frame_box_cv = inference(frame_cv)
    print "Inference time: {}".format(time.time() - t)

    cv2.imshow("window", frame_box_cv)
    cv2.waitKey(3)

    frame_box_ros = bridge.cv2_to_imgmsg(frame_box_cv, "bgr8")
    pub.publish(frame_box_ros)

#Creamos la sesión en TensorFlow

sess = tf.Session(config = config)

#Publicamos el tópico /frame_box

pub = rospy.Publisher('frame_box', Image, queue_size=10)

#Nos subscribimos al tópico 7frame_descomprimido para tomar la imagen que procesaremos con la red
neuronal

sub = rospy.Subscriber('frame_descomprimido', Image, callback)

# Evita que python salga hasta que se detiene este nodo

rospy.spin()

# Cerramos sesión

sess.close()
```