Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Detección y tracking de objetos mediante la Mask R-CNN

Autor: Javier García Fontán

Tutor: Rafael Barea Navarro

2019

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Detección y tracking de objetos mediante la Mask R-CNN

Autor: Javier García Fontán

Tutor: Rafael Barea Navarro

Tribunal:

Presidente: Marta Marrón Romera

Vocal 1º: Javier Macías-Guarasa

Vocal 2º: Rafael Barea Navarro

Fecha: 18 de septiembre de 2019

Índice general

Índice de figuras

Índice de tablas

Resumen

En este trabajo, se va a realizar la configuración de un sistema embebido, que irá dentro del coche autónomo del proyecto *"Smart Elderly Car: The UAH Intelligent Electric Vehicle"* y se encargará del procesado de imagen. Se dejará configurado en un estado por defecto, pero compatible con los distintos módulos del proyecto, donde si un integrante del proyecto ha mejorado la red que se encarga del procesado de imagen, se puede integrar facilmente en el sistema siguiendo el manual de usuario que se describe más adelante.

Palabras Clave: Sistema Embebido, Jetson AGX Xavier, Cámara ZED, Ros, Red Neuronal Convolucional.

Abstract

In this work, the configuration of an embedded system is going to be carried out, which will go inside the autonomous car of the *"Smart Elderly Car: The UAH Intelligent Electric Vehicle"* proyect and will be in charge of the image processing. Will be left configured in a default state, but compatible with the different modules of the project, where if a member of the project has improved the network that is responsible for image processing, can be easily integrated into the system following the user manual described below.

Keywords: Embedded System, Jetson AGX Xavier, ZED Camera, Ros, Convolutional Net Network.

Capítulo 1

Introducción

1.1 Presentación

La Organización Mundial de la Salud, estima que para el 2030, casi un tercio de la población mundial, vivirá en las ciudades y ya en el 2011 la Comisión Europea indicaba que deberían de proponerse nuevas formas de movilidad que aporten soluciones sostenibles para la seguridad de las personas y las mercancías. En cuanto a la seguridad, se busca reducir al mínimo el numero de muertes en accidentes de tráfico, pero hoy día, ese número no es bajo, solo en 2014 más de 25.700 personas murieron en las carreteras de las Unión Europea y algunos estudios muestran que los accidentes fatales aumentan con la edad de las personas de 65 años o más.

En los últimos años, numerosas universidades, empresas privadas e incluso las industrias automovilísticas, ven la **conducción autónoma** como uno de los pilares fundamentales en la resolución de los problemas de tráfico en las zonas urbanas y a su vez, uno de los grandes retos de de la automoción actual. La existencia de vehículos autónomos fiables y economicamente asequibles tendrán un gran impacto en la sociedad y que afectará a los aspectos medioambientales, demográficos, sociales y económicos. Se estima que reducirá las muertes en carretera, mejorará el flujo del tráfico, reducirá las emisiones nocivas y mejorará la movilidad de personas con facultades deficientes, como las personas de edad avanzada o personas con discapacidad. Y aunque las predicciones, en cuanto a la circulación real de vehículos autonomos, sean muy optimistas, la conducción en entornos urbanos llevará mas tiempo, que la conducción en autopistas, debido a su complejidad e incertidumbre.

Siguiendo esta línea, el **grupo de investigación Robesafe** [3] de la **Universidad de la Alcala** está desarrollando el proyecto *"Smart Ederly Car"* [4], que consiste en el desarrollo de un vehículo eléctrico y autónomo, y esta orientado a la circulación en zonas urbanas para la población de edad avanzada. Ya existe un primer prototipo de vehículo operativo y diseñado al 100 % en la Universidad de Alcalá (Figura 1.1). Este TFG se centra en una de las etapas de percepción del entorno que rodea al vehículo, en concreto, en la preparación y configuración del sistema embebido encargado del procesado de imagen RGB.

Figura 1.1: Prototipo de de la UAH.

1.2 Objetivos del Proyecto

El TFG continua la línea de desarrollo que realizó un antiguo alumno de la UAH, en su TFG [1], donde desarrollo un sistema de detección y tracking de objetos empleando técnicas de deep learning con imagenes RGB y fusión de datos con los resultados de un láser 3D. Ese sistema no se llego a utilizar en un entorno real, aunque se validó los resultados con la base de datos *The KITTI Vision Benchmark Suite* [2] que provee de una gran cantidad de información, como imagenes, datos de calibración o medidas de nubes de puntos y toda esta información son de datos reales, por lo que se puede decir que al menos el sistema se validó con datos fiables.

Así que, este trabajo continuará este desarrollo, donde se instalará en un sistema embebido diseñado para el procesado de imagen para su uso en el entorno real y poder validarlo. Este sistema embebido, habrá que configurarlo para integrarlo con el sistema global (Figura 1.2). En resumen, las tareas a realizar son:

- Recuperar el sistema de detección y tracking del alumno anterior y optimizarlo en la medida de lo posible.
- Poner a punto el sistema embebido, al que se le instalará:
  - Drivers de la cámara por la que se obtendrá las imágenes RGB del entorno.
  - ROS, ya que será el medio de comunicación entre los módulos del sistema.
  - Integrar el sistema de detección del punto anterior
- Testear su rendimiento.

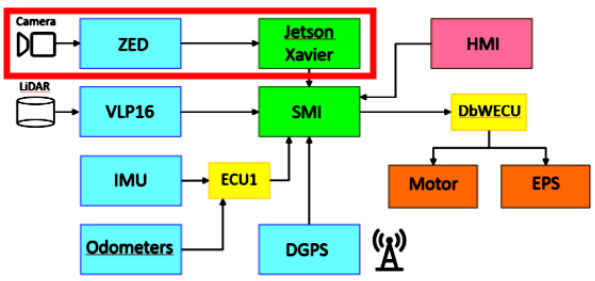


Figura 1.2: Arquitectura del sistema. Remarcado en rojo el módulo realizado en el TFG.

Capítulo 2

Estado del Arte

Actualmente no hay muchas empresas que desarrollen sistemas embebidos destinados al uso del procesamiento de imagen mediante la GPU, pero al menos, la empresa NVIDIA sí ha desarrollado, y aunque es una de las principales proveedoras del mercado tecnológico, desde 2014 se ha centrado en cuatro bloques que son los videojuegos, centros de datos, el avance de dispositivos autónomos y la visualización computacional.

En el desarrollo de este último, Nvidia ha creado el catálogo de **Jetson**, cuyos módulos incluyen un gran abanico de aplicaciones que requieren varios niveles de rendimiento y con distintos precios. Los módulos de Jetson cuentan con un rendimiento y eficiencia energética en un factor de forma pequeño, lo que proporciona de forma efectiva la potencia de la IA moderna, deep learning y la inferencia a sistemas integrados perimetrales.

Jetson Nano

Con tan solo 70 x 45 mm, el módulo Jetson Nano es el dispositivo Jetson más pequeño. Proporciona 472 GFLOP para ejecutar los algoritmos de la IA moderna de forma rápida. Ejecuta varias redes neuronales en paralelo y procesa varios sensores de alta resolución simultáneamente, lo que lo hace idóneo para aplicaciones como grabadores de vídeo de red (NVR) básicos, robots domésticos y gateways inteligentes con completas funciones de análisis con consumos de entre 5 y 10 vatios solamente. Tiene un precio de 109 euros. (Tabla de especificaciones técnicaa en el apendice B.1).

Serie Jetson TX2

Experimenta más del doble de rendimiento o el doble de eficiencia energética de Jetson TX1. Todo esto es posible gracias a la arquitectura NVIDIA Pascal de 256 núcleos y la memoria de 8 GB de Jetson TX2 que se traducen en el cálculo y la inferencia más rápidos. Puede ejecutar grandes redes neuronales profundas para conseguir la máxima precisión en dispositivos perimetrales. Con solo 7,5 watts, ofrece 25 veces más eficiencia energética que una CPU de sobremesa de última generación. Esto lo convierte en la opción perfecta de procesamiento en tiempo real en aplicaciones en las que el ancho de banda y la latencia pueden ser un problema. Entre estas aplicaciones se incluyen robots de fábrica, drones comerciales, dispositivos de colaboración empresarial y cámaras inteligentes para ciudades inteligentes. Tiene un precio de 419 euros. (Tabla de especificaciones técnicaa en el apendice B.2).

Serie Jetson Xavier

Jetson AGX Xavier ofrece un gran rendimiento de estación de trabajo a 1/10 del tamaño de una estación de trabajo. Esto lo convierte en la opción perfecta para máquinas autónomas tales como

robots de logística y reparto, sistemas de fábrica y grandes aviones no tripulados para la industria. El alto rendimiento de Jetson AGX Xavier puede manejar algoritmos de odometría visual, fusión de sensores, localización y cartografía digital, detección de obstáculos y planificación de rutas decisivos para los robots de nueva generación. Consigue el rendimiento de estación de trabajo con GPU de 32 TeraOPS (TOPS) de cálculo máximo y 750 Gbps de E/S de alta velocidad sin precedentes en un compacto formato de 100 x 87 mm. Los usuarios pueden configurar modos operativos a 10 W, 15 W y 30 W según sea necesario para sus aplicaciones, lo que permite nuevos niveles de densidad de cálculo, eficiencia energética y funciones de inferencia de IA en primera línea. Tiene un precio de 838 euros. (Tabla de especificaciones técnica en el apendice B.3).

Capítulo 3

Desarrollo

El sistema embebido elegido es la **NVIDIA Jetson AGX Xavier** por su potencia frente al resto de sistemas y será el encargado de recibir la imagen RGB de la cámara **ZED**[1] y de ejecutar al sistema de detección de objetos y tracking en tiempo real.

También se instalará **ROS**[2]ya que sistema del vehículo sigue una arquitectura modular en la que los módulos individuales procesan la información de forma asíncrona y estos módulos se comunican entre sí mediante el sistema de comunicación entre procesos, en particular, el paradigma de publicar-subscribir se utiliza para proporcionar comunicaciones sin bloqueo. Cada módulo corresponde a un proceso Linux independiente que se ejecuta en diferentes procesadores.

A continuación, se expondrá la arquitectura y funcionamiento del sistema de detección y la instalación y configuración de la Jetson Xavier.

3.1 Mask R-CNN

Lo primero es recuperar el sistema de detecciones y tracking del alumno anterior. El sistema está montado en un **docker**, lo que hace muy fácil su portabilidad entre distintos terminales, al igual que el sistema de validación, **kitti**.

3.1.1 Arquitectura del sistema

La arquitectura emplea una **CNN** [3], la **Mask R-CNN**, que detecta los objetos de la imagen RGB de entrada y los encierra en una 'bounding box' [4] y hace la segmentación semántica de esos objetos. En este punto se bifurca el trabajo, por un lado, se guarda la detecciones de la imagen 2D, y por otro lado, con la segmentación semántica de la escena, técnicas de procesado de imagen y fusión de datos de la información obtenida por la nube de puntos del **LIDAR**[5], se obtiene la segmentación semántica de la imagen en 3D y las 'bounding box' de las detecciones, también en 3D. Pero las detecciones calculadas de la escena en 3D tienen falsos positivos, por lo que aprovechando la buena precision de la Mask R-CNN, se comparan las detecciones del 2D frente a la otra y solo se mantienen las detecciones que coinciden con las detecciones 2D. Dicha arquitectura se muestra en la figura 3.1.

[1] Cámara estereoscópica
[2] Robotic Operation System
[3] Convolutional Net Network
[4] Area mínima que contiene al objeto
[5] Sensor laser que mide distancias en el espacio. Tiene un campo de visión horizontal de 360º y vetical de +- 15º

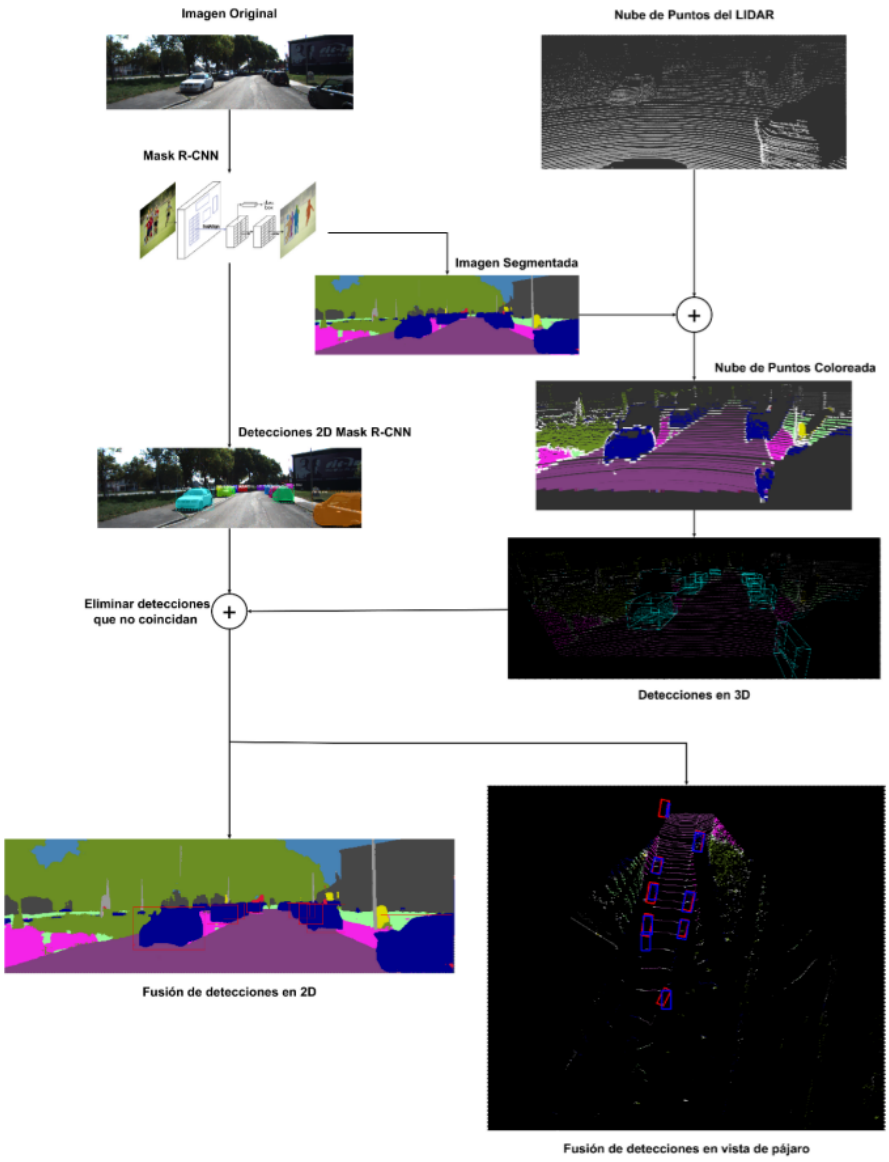


Figura 3.1: Arquitectura del sistema inicial

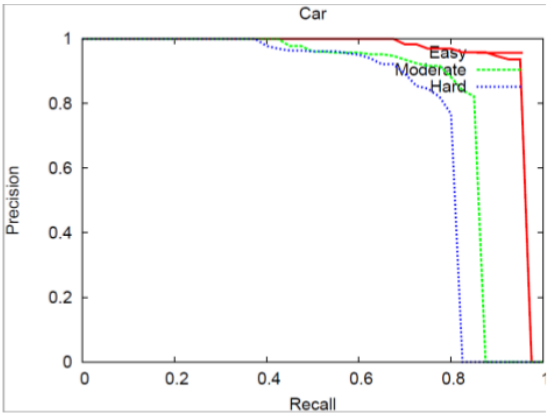
3.1.2 Resultados

La propia base de datos de Kitti, obrece un set para testear los resultados del sistema. El formato de resultados que nos muestra el test está dividido en tres modalidades, que son: **easy**, **moderate** y **hard**. Estas modalidades se refieren a la dificultad que existe a la hora de detectar los coches que pertenecen a cada una, de manera que un coche que se encuentre a una distancia muy grande pertenecerá a la modalidad **hard**, mientras que uno que se encuentre en un punto cercano y donde tengamos una buena visibilidad del mismo pertenecerá a la modalidad **easy**.

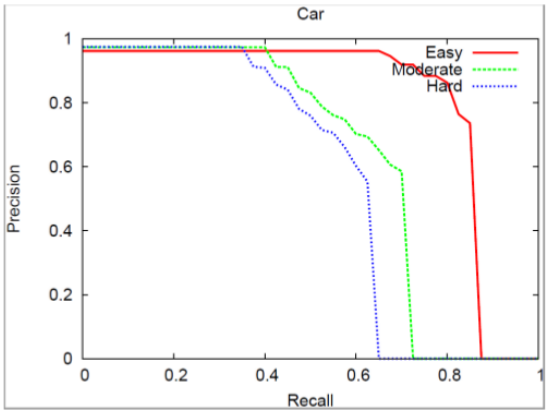
Los resultados serán representados de dos formas diferentes:

- **En forma de gráfica**: Se mostrará una gráfica en la que aparecerán tres líneas de resultados diferentes, correspondiendo cada uno de ellos con las tres dificultades de detección antes mencionadas. La gráfica será representada en base a dos parámetros, la precisión de las detecciones y un parámetro llamado recall. El parámetro de precisión hace referencia a lo bien que se ajustan los

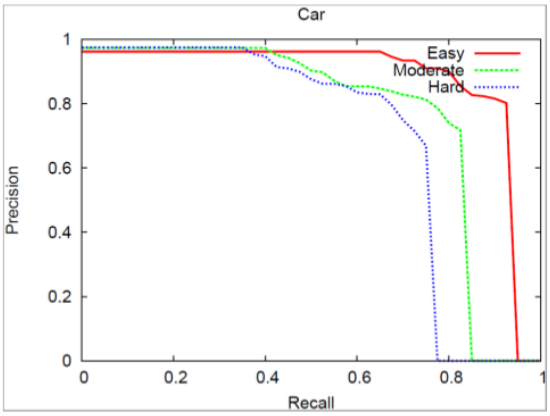
resultados de las cajas que hemos detectado con las cajas mínimas reales que envolverían a los coches que se encuentran en los ficheros de ground truth que nos proporciona la base de datos. El segundo parámetro está relacionado con la cantidad de objetos detectados y los falsos positivos, de manera que este parámetro se verá penalizado si no detectamos como objetos algunos de los que si que están determinados en los ficheros de ground truth o si cometemos errores indicando que hay coches donde realmente no los hay (Figura 3.2).



(a) Resultados 2D



(b) Resultados 3D



(c) Resultados de vista de pájaro

Figura 3.2: Resultados gráficos del sistema con la Mask R-CNN al aplicar el banco de pruebas de Kitti

- **En forma de numérica**: Esta segunda forma de representar los resultados será en realidad el área bajo la curva de la gráfica con la que se representan los datos. De esta manera tendremos una idea más exacta del conjunto de coches detectados y la precisión de las detecciones que hemos realizado (Tabla 3.1).

Entrenamiento	Dificultad		
	Easy	Moderate	Hard
Detección 2D	0.9407	0.8253	0.7715
Detección 3D	0.8024	0.6243	0.5593
Vista pájaro	0.8693	0.7562	0.6863

Tabla 3.1: Resultados númericos del sistema con la Mask R-CNN al aplicar el banco de pruebas de Kitti

Los resultados son bastante buenos en general, sobre todo en imágenes 2D y decae algo más en imagenes 3D, pero sin bajar del 50 %.

Todos estos resultados se centran en la calidad, en sí se detecta correctamente o no, pero hay un punto que no se tuvo en cuenta y si ha supuesto un problema para este trabajo, la **velocidad de procesado**.

Solamente en la fase de segmentación de la Mask R-CNN, ejecutandose en un ordenador medio, tarda 20 segundos en obtener la imagen segmentada y en el ordenador de trabajo que tiene buenas características tarda 4,5 segundos. Estos resultados hacen que no se pueda usar esta arquitectura, ya que el sistema iría en un vehículo autónomo y las restricciones de funcionamiento en tiempo real lo hace inviable.

3.2 Alternativa a la Mask R-CNN

Ya que el problema está en la Mask R-CNN, no por su funcionamiento sino por su tiempo de procesado, se propuso el uso de otra CNN que sustituya el funcionamiento de detección de objetos, la **YOLO**. Esta CNN es conocida por su velocidad y su función en la detección de objetos. Siguiendo la arquitecturá del sistema que empleaba la Mask R-CNN, es necesario una CNN que haga la segmentación semántica de la escena, y esa es la **ERFNet**, que obtiene una segmentación semantica 2D en tiempo real.

Al final, la lógica de la arquitectura es la misma (Figura 3.3), pero para asegurarse de que esta arquitectura si funcione en tiempo real, solo se integrará la recepción de imagen RGB y la detección de objetos 2D con la YOLO en la Jetson Xavier. La parte de la segmentación semántica, la fusión de datos con la información del lidar para finalmente obtener una detección de objetos en 3D y el tracking de los objetos, se hará en un ordenador exterior, concretamente en el msi que aparece en la figura 1.2 de la página 2.

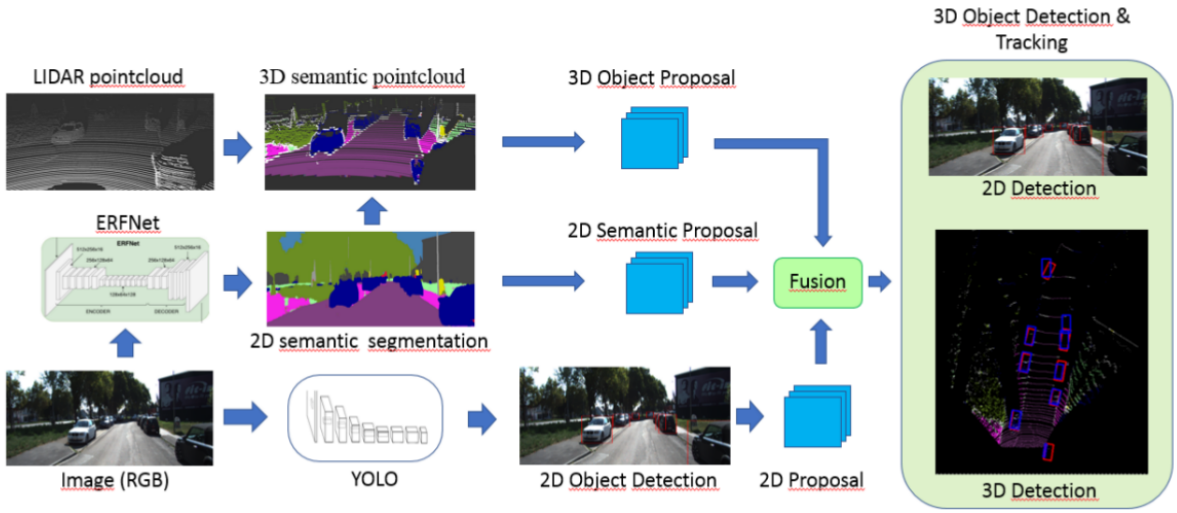


Figura 3.3: Arquitectura de las CNN. YOLOv3-ERFNet

3.3 Instalación del entorno de la Jetson

A continuación se procederá a exponer todos los pasos llevados para tener operativa la Jetson.

3.3.1 Introducción

Para trabajar con la Jetson, Nvidia provee del instalador **Nvidia Jetpack SDK**, que se compone de una serie de componentes para el desarrollo de diversas aplicaciones. Usa NVIDIA SDK Manager para

actualizar la Jetson con la última imagen del Sistema Operativo, instala herramientas de desarrollo tanto para el host como la propia Jetson, instala la librerias, interfaces multimedia, ejemplos y documentacion necesaria para impulsar el entorno de desarrollo.

Al instalar Nvidia Jetpack SDK se puede instalar todo lo siguiente:

OS Image.
Incluye un sistema de ficheros derivado de Ubuntu.

Librerias.
Incluyes las siguientes librerias:

- **Tensort** y **cuDNN** para aplicaciones de deep learning
- **CUDA** para acelerar aplicaciones con la GPU
- Paquetes de **Interfaces Multimedia** para aplicaciones con camaras o desarrollo de drivers de sensores
- **VisionWorks** y **OpenCV** para aplicaciones de computación visual

Herramientas de desarrolladores
Incluyes las siguientes herramientas

- **Herramientas de CUDA:** Nsight Eclipse Edition IDE, herramientas de depuracion y de creacion de perfiles y una serie de herramientas para la compilar de forma cruzada aplicaciones aceleradas por GPU.
- **NVIDIA Nsight System:** Un rastreador del sistema y analizador multi-nucleo de la CPU del PC que proporciona una vista de los datos capturados, lo que ayuda a mejorar el rendimiento general de la aplicacion.
- **NVIDIA Nsight Graphics:** Una herramienta de nivel de consola que permite a los desarrolladores depurar y perfilar OpenGL, OpenGL ES y Vulkan.

3.3.2 Instalar Jetpack

3.3.2.1 Descargar e instalar JetPack

Se tiene que descargar el instalador de Jetpack [6] [6] en un ordenador con Linux 16.04 o posterior para que haga de host e instalarlo. Al darle a descargar, la página de Nvidia nos pide que nos registremos en el programa de desarrolladores de Nvidia (Figura 3.4), por lo que habrá que registrarse e iniciar sesión (Figura 3.5) y ya se podrá descargar el instalador(Figura3.6)

Despues de la instalación, se puede ir al buscador del equipo y poner *sdkmanager* para encontrar el instalador(Figura 3.7).

3.3.2.2 Configurar la descarga

Se ejecuta el instalador y de nuevo pedirá una cuenta en el equipo de desarrolladores de Nvidia, pero como ya se creo una en el paso anterior, se introduce la misma (Figura 3.8)

A continuación, se elegirá el modelo de la **Jetson AGX Xavier** [7](Figura 3.9)

[6]Se ha descargado la última versión disponible a fecha de la realización del trabajo, la version 4.2.
[7]Las imagenes del proceso de flasheo se corresponden con el modelo TX2 ya que es donde se hicieron las primeras pruebas, pero el proceso es identico en ambas placas a excepción de la selección del hardware

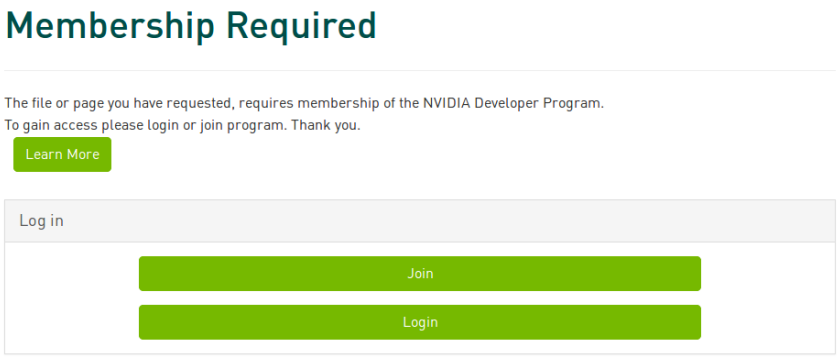


Figura 3.4: Unirse al programa de desarrolladores de Nvidia

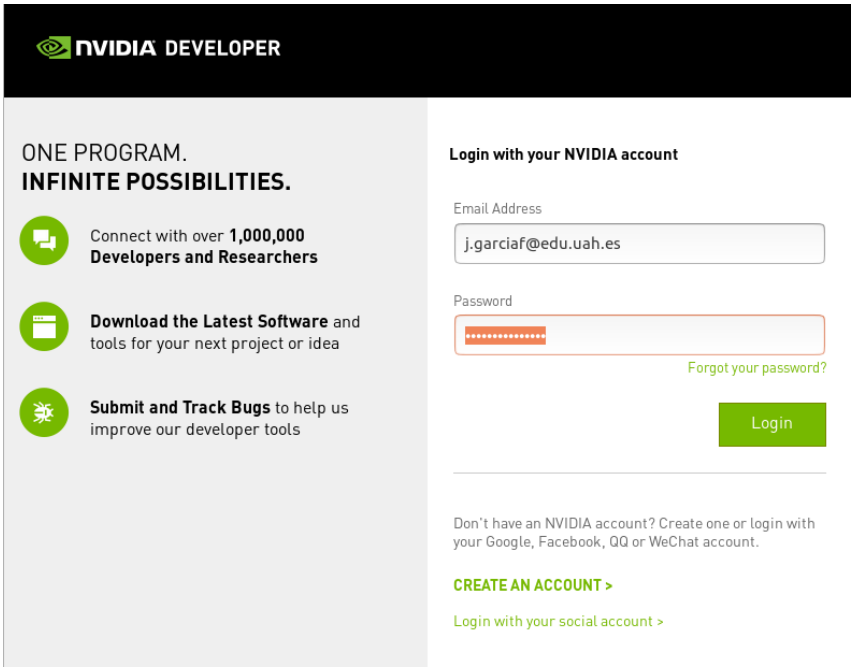


Figura 3.5: Iniciar sesión en el programa de desarrolladores de Nvidia

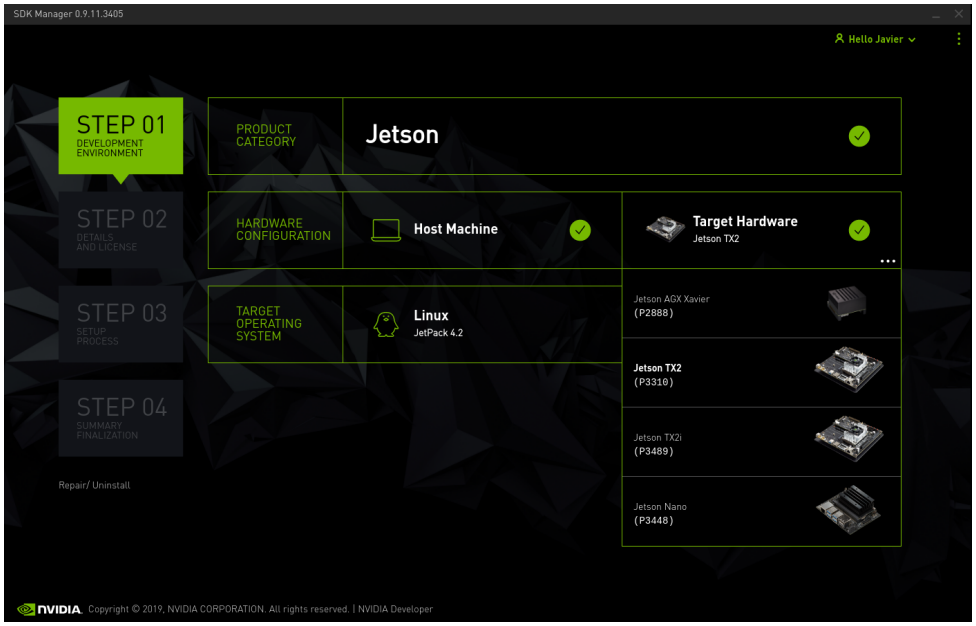


Figura 3.9: Elección del modelo de la Jetson

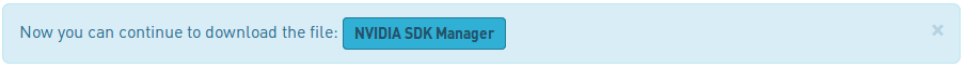


Figura 3.6: Acceso a NVIDIA SDK Manager

Figura 3.7: Localización del SDK Manager en el host

Lo que viene a continuación es la selección del software que se quiere instalar en la Jetson y para ello, solamente hay que marcar las casillas que se corresponden con el software elegido y se da a descargar (Figura 3.10). Este proceso solo descargar el software en el ordenador que hace de host y lo prepara para el momento del flasheo en la Jetson. Una vez se haya descargado todo lo necesario saldra algo parecido a la figura 3.11.

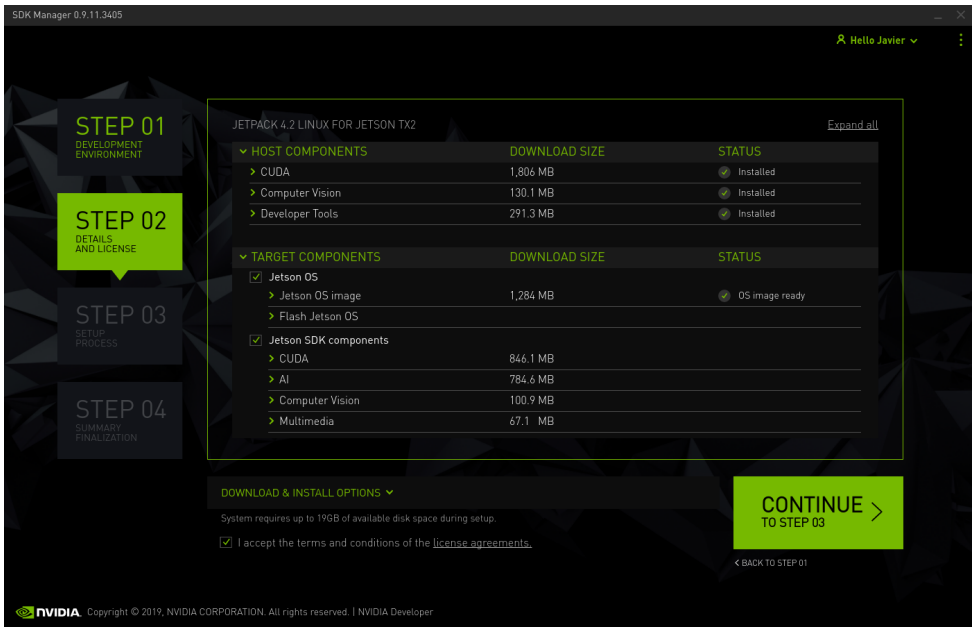


Figura 3.10: Elección del software de la Jetson

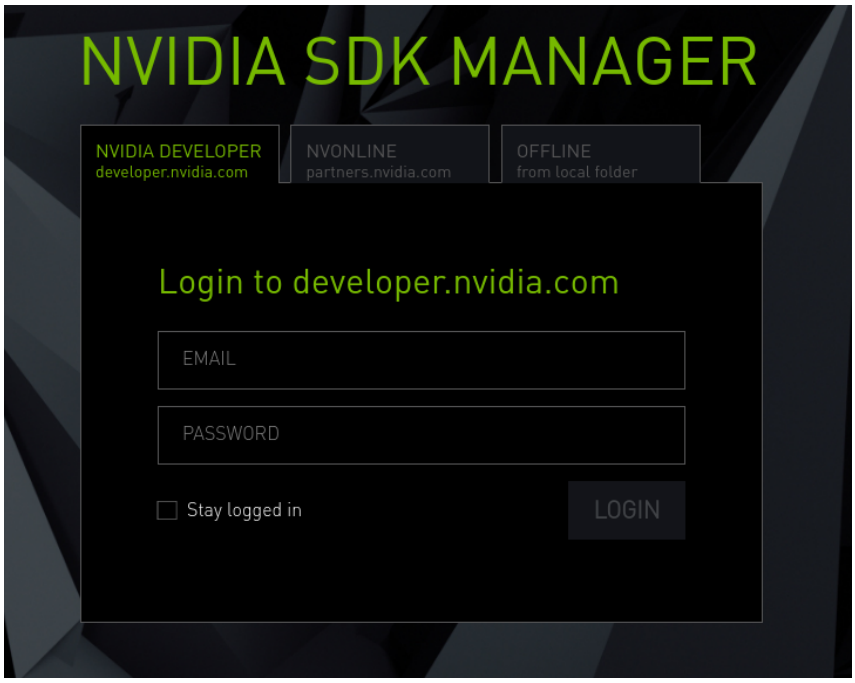


Figura 3.8: Iniciar Sesión en el Nvidia SDK Manager

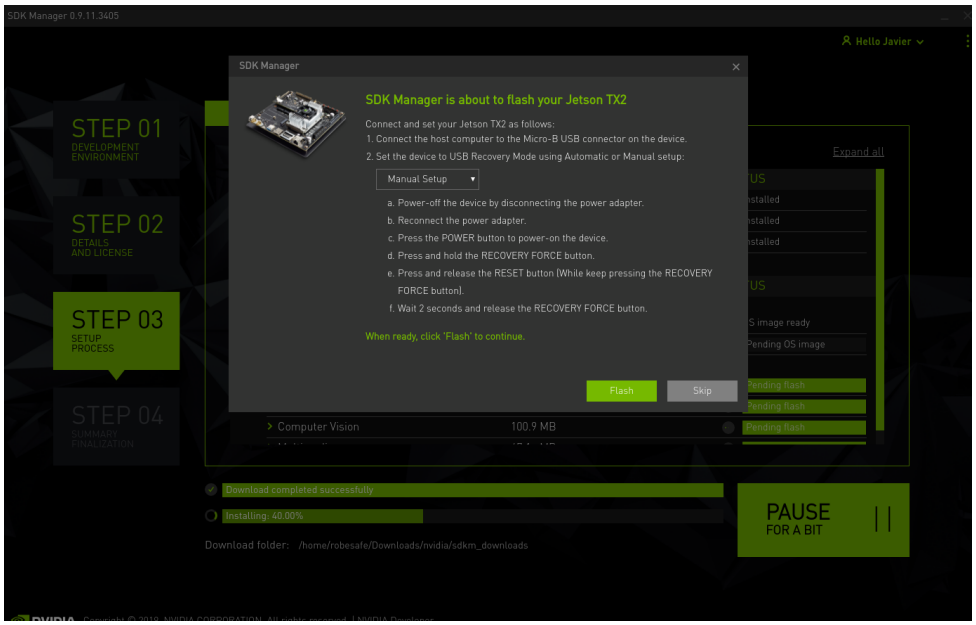


Figura 3.11: Descarga del software finalizada. Iniciar el flasheo de la Jetson

3.3.2.3 Modo Recovery de la Jetson

Ahora es necesario conectar la Jetson al host y configurarla en modo recovery para poder flashearla.

- Conectar la Jetson al host.

  1. Utiliza el cable usb tipo C + tipo A que trae el kit de la Jetson y conecta el conector C a la Jetson y el tipo A al host.
  2. Conecta un monitor a la Jetson mediante un cable HDMI.

3. Conectar un teclado y raton a la Jetson.
4. Conecta la fuente de alimentación que incluye el kit de la Jetson (Deberia de estar apagada.)

- Modo recovery

1. Conectar la Jetson al host (punto anterior).
2. Pulsar el botón 'Power' para encender la Jetson.
3. Mantener pulsado el botón 'Force Recovery'.
4. Sin soltar el botón anterior, mantener pulsado del botón 'Reset's.
5. Mantener unos segundos y soltar.

3.3.2.4 Flashear e instalar el software en la Jetson

Si se ha seguido correctamente los pasos, al darle al botón de 'Flash', aparecerá otra ventana como la de la Figura 3.12, donde solicita el nombre de usuario y contraseña que tendra la Jetson, pero todavía no hay que tocar nada aquí porque hay que esperar a que se instale el Sistema Operativo.

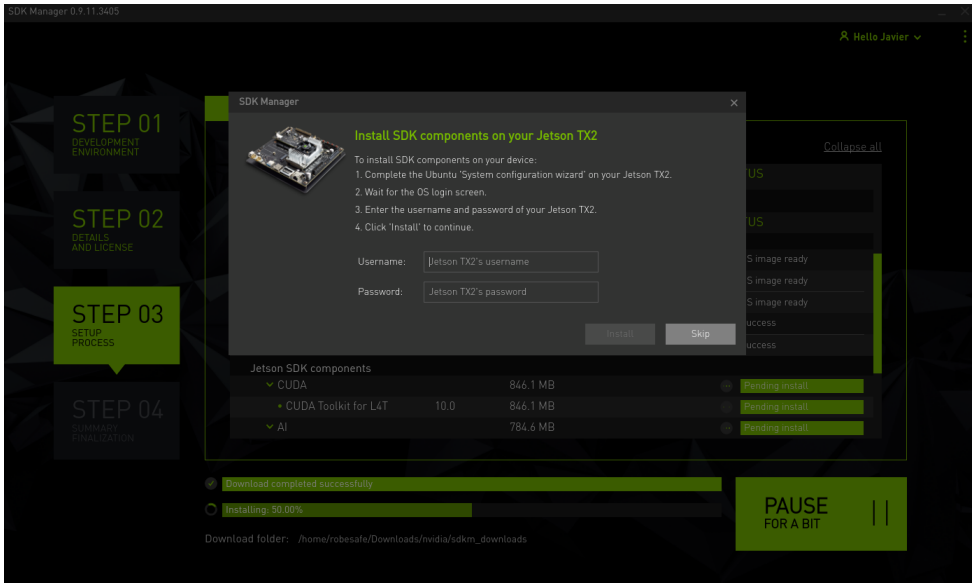


Figura 3.12: Flasheo finalizado. Instalar el software en la Jetson

La primera parte de la instalación dura aproximadamente unos 15 minutos y luego aparecerá una ventana emergente y notificará que el resto de la instalación continuara desde la Jetson. Esta parte de la instalación es similar a cualquier otra distribución de Linux, donde soliciará aceptar los terminos de instalación, selección de idioma, tipo de teclado, localización y pedirá el nombre usuario y contraseña que tendra la Jetson.

Usuario: robesafe

Equipo: TX2 [8]

Contraseña: robesafe

[8] En la jetson AGX Xavier, el nombre de equipo es 'Xavier'

Se espera a que instale el Sistema Operativo y a que la propia Jetson se reinicie para aplicar los cambios. Cuando aparezca la ventana de inicio de sesión, se deja en ese estado y se continua desde el host, donde solicita las credenciales de la Jetson para instalar los paquetes restantes y se le da a instalar (Figura 3.12).

Se espera que se instalé todo y con esto ya esta la Jetson preparada para ejecutarse sin problemas.

3.4 Ros

Para establecer una comunicación entre los distintos módulos del proyecto se empleará el framework de **ROS** [9], que nos permitirá **publicar** los datos en una serie **topics** que podrán ser leídos por cualquier otro módulo que se **subscriba** a ellos.

Hay una versión de ROS distinta por cada versión de Ubuntu y como la versión de Ubuntu instalada se corresponde con la 18.04, entonces se instalará el **ROS Melodic Morenia** [8]

3.4.1 Instalar ROS Melodic Morenia

3.4.1.1 Instalar ROS desde el terminal.

Añadir los paquetes de ROS al sistema.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc)
main" > /etc/apt/sources.list.d/ros-latest.list'
```

Actualizar las claves.

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key
C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Actualizar los paquetes.

```
sudo apt update
```

Instalar ROS, la versión recomendada de escritorio.

```
sudo apt install ros-melodic-desktop-full -y
```

Inicializar el rosdep

```
sudo rosdep init
rosdep update
```

Configurar el entorno de trabajo

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Instalar dependencias para la creacion de paquetes

```
sudo apt install python-rosinstall python-rosinstall-generator python-wstool
build-essential
```

[9] Robotic Operation System

3.4.1.2 Preparar el entorno desde el terminal.

Creacion y configuracion del entorno de trabajo. Se localizara en '/home/robesafe' y se llamara 'catkin_ws', pero se puede llamar como se quiera, solamente habrá que sustituir 'catkin_ws' por el nombre que se prefiera del código de abajo.

```
mkdir -p ~/catkin_ws/scr
cd ~/catkin_ws/scr
catkin_init_workspace
cd ..
catkin_make
```

Al final, la estructura del entorno de trabajo debería ser como la de la figura 3.13.

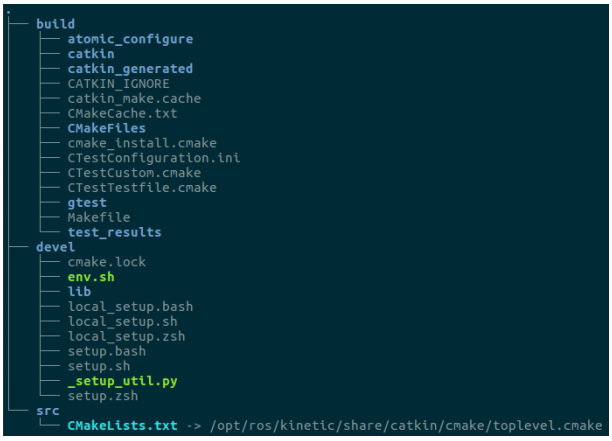


Figura 3.13: Arbol de ficheros del entorno de trabajo de ROS

Configurar variables y rutas de entorno

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
echo "export ROS_WORKSPACE=~/catkin_ws/" >> ~/.bashrc
echo "export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$ROS_WORKSPACE" >> ~/.bashrc
```

Configurar las direcciones de red. Ahora mismo, la dirección de red puesta es la del localhost, pero más adelante se tendrá que cambiar para que los distintos módulos del sistema se puedan comunicar entre sí.

```
echo "export ROS_MASTER_URI=http:\\127.0.0.1:11311" >> ~/.bashrc
echo "export ROS_IP=127.0.0.1" >> ~/.bashrc
```

Cargar el bashrc

```
source ~/.bashrc
```

3.5 ZED

La ZED es una cámara 3D para la detección de profundidad, seguimiento de movimiento y mapeo 3D en tiempo real.

En la página oficial de ROS, tienen disponible un paquete que permite comunicar la cámara con ROS y así publicar la información que obtiene la cámara [11].

3.5.1 Instalar ZED

Para trabajar con la ZED se necesita tener instalado:

- Ubuntu 16.04 o superior
- CUDA
- ROS Kinetic o superior
- ZED SDK 2.3 o superior

Ya esta todo instalado, excepto el ZED SDK, que en la propia página de Stereolabs se facilita el software [10][10].

Al descargar el instalador, hay que darle permisos de ejecución y se ejecuta.

```
sudo chmod +x ZED_SDK_JP4.2_v<version_descargada>.run
./ZED_SDK_JP4.2_v<version_descargada>.run
```

Se acepta todo y ya esta instalada la dependencia de CUDA. Ahora quedaría descargar el paquete de ROS y compilarlo.

```
cd ~/catkin_ws/src
git clone https://github.com/stereolabs/zed-ros-wrapper.git
cd ..
catkin_make
rospack profile
```

Para probar que se ha compilado correctamente el paquete de ROS se puede lanzar la cámara sin rviz[11]

```
roslaunch zed_wrapper zed.launch
```

Y con rviz

```
roslaunch zed_display_rviz display.launch
```

3.6 YoloV3

Yolo[12][12] es un sistema de detección de objetos en tiempo real y su implementación 'oficial' es **Darknet**, que es un framework de redes neuronales escritas en C con CUDA, por lo que soporta tanto el cómputo en CPU como en GPU.

3.6.1 Instalar YoloV3

Ya existe un paquete en ROS que implementa el framework Darknet[13] y con los siguientes comandos se descargará y compilará. Recordar que hay que cambiar el nombre del espacio de trabajo de ros si no se uso 'catkin_ws' por el nombre que se haya puesto.

[10] La versión del ZED SDK instalada es la 2.8

[11] Es un visualizador 3D que se instaló al instalar ROS (debido al paquete de escritorio) y muestra los datos de los sensores conectados a ROS

[12] You Only Look Once

```
cd ~/catkin_ws/src
sudo git clone --recursive https://github.com/leggedrobotics/darknet_ros.git
cd ..
catkin_make -DCMAKE_BUILD_TYPE=Release
```

3.6.2 Configurar el paquete de Ros Darknet

Es necesario crear un tipo de mensaje que sea común para todas las partes del proyecto y, obviamente, el paquete descargado no lo trae por defecto ya que es un convenio entre los distintos miembros del proyecto. Así que, hay que crear un tipo de mensaje específico, modificar el nodo del paquete para integrar el nuevo tipo de mensaje y leer de unos de los topic que publica la camara ZED.

3.6.2.1 Crear los mensajes publicados por el topic del nodo Darknet

Estos mensajes (Figura 3.14a y 3.14b) se crearán en la ruta '$ROS_WORKSPACE/src/darknet_ros/darknet_ros_msgs/msg' y en nivel inferior de ese directorio esta el fichero 'CMakeLists.txt', en el cual hay que añadir los nombres de los mensajes creados (Figura 3.14c).

yolo_obstacle.msg

```
string type
float64 probability
float64 x1
float64 y1
float64 x2
float64 y2
float64 h
float64 w
float64 l
float64 tx
float64 ty
float64 tz
```

(a) Información de cada detección

yolo_list.msg

```
Header header
Header image_header
yolo_obstacle[] yolo_list
```

(b) Conjunto de detecciones en una sola interacción

```
add_message_files(
  FILES
    BoundingBox.msg
    BoundingBoxes.msg
    yolo_list.msg
    yolo_obstacle.msg
)
```

(c) Tipo de mensajes añadidos al nodo

Figura 3.14: Mensaje del nodo de Ros con la información de la Darknet

3.6.2.2 Capturar la imagen de la ZED

La imagen que lee el nodo Darknet puede provenir de un fichero o leerlo de un topic y como la ZED tiene sus propios nodos publicando la información que recibe, entonces se aprovechará de esta situación y se modificará el nodo de la Darknet para que se subscriba a uno de los topics. Así que, cuando la Darknet termine de procesar una imagen y publicar sus resultados, recibiría constantemente imagenes nuevas y así sucesivamente.

El fichero a modificar se encuentra en la ruta '$ROS_WORKSPACE/src/darknet_ros/darknet_ros/src/' y se llama '**YoloObjectDetector.cpp**'. Hay que cambiar la línea 143 de la figura 3.15a por la línea 148 de la figura 3.15b. Con esto, siempre que se ejecute el nodo de la Darknet, va a subscribirse a ese topic.

```
141
142   nodeHandle_.param("subscribers/camera_reading/topic", cameraTopicName,
143                     std::string("/camera/image_raw"));
144   nodeHandle_.param("subscribers/camera_reading/queue_size", cameraQueueSize, 1);
```

(a) Código Original

```
146
147   nodeHandle_.param("subscribers/camera_reading/topic", cameraTopicName,
148                     std::string("/zed_node/left/image_rect_color "));
149   nodeHandle_.param("subscribers/camera_reading/queue_size", cameraQueueSize, 1);
```

(b) Código modificado

Figura 3.15: Subscripción al topic de la ZED

3.6.2.3 Crear el topic con el nuevo mensaje

Ahora hay que crear el topic que publique la información procesada por la Darknet. Desde el mismo fichero del apartado anterior, hay que inicializar el publicador, así que, al final del bloque donde se inicializan los subscriptores y publicadores (Figura 3.16a), se añade el publicador deseado (Figura 3.16b).

```
129   // Initialize publisher and subscriber.
130   std::string cameraTopicName;
131   int cameraQueueSize;
132   std::string objectDetectorTopicName;
133   int objectDetectorQueueSize;
134   bool objectDetectorLatch;
135   std::string boundingBoxesTopicName;
136   int boundingBoxesQueueSize;
137   bool boundingBoxesLatch;
138   std::string detectionImageTopicName;
139   int detectionImageQueueSize;
140   bool detectionImageLatch;
```

(a) Código Original

```
139   std::string yoloListTopicName;
140   int yoloListQueueSize;
141   bool yoloListLatch;
```

(b) Código añadido

Figura 3.16: Inicialización del publicador

En el bloque donde se crean los nodos (Figura 3.17a), se añade al final de estos el nodo que se necesita (Figura 3.17b).

```
155   nodeHandle_.param("publishers/detection_image/queue_size", detectionImageQueueSize, 1);
156   nodeHandle_.param("publishers/detection_image/latch", detectionImageLatch, true);
```

(a) Código Original

```
157   nodeHandle_.param("publishers/yolo_list/topic", yoloListTopicName,
158                     std::string("yolo_list"));
```

(b) Código añadido

Figura 3.17: Creación del nodo

En el bloque donde se asignan los subscriptores o publicadores a un nodo (Figura 3.18a), se añade al final de estos el publicador que se necesita (Figura 3.18b).

```
163   boundingBoxesPublisher_ = nodeHandle_.advertise<darknet_ros_msgs::BoundingBoxes>(
164       boundingBoxesTopicName, boundingBoxesQueueSize, boundingBoxesLatch);
165   detectionImagePublisher_ = nodeHandle_.advertise<sensor_msgs::Image>(detectionImageTopi
166                                                                        detectionImageQueu
167                                                                        detectionImageLatc
```

(a) Código Original

```
176   yoloListPublisher_ = nodeHandle_.advertise<darknet_ros_msgs::yolo_list>(
177       yoloListTopicName, yoloListQueueSize, yoloListLatch);
```

(b) Código añadido

Figura 3.18: Creación del publicador

En el bloque donde se rellenan los resultados en el mensaje (Figura 3.19a), se añade al final de estos los datos nuevos del nuevo mensaje (Figura 3.19b). Hay datos que se rellenan con 0, esto es porque el nodo Darknet no utiliza esos campos pero deben de existir para mantener el formato de mensaje.

```
614             boundingBox.Class = classLabels_[i];
615             boundingBox.probability = rosBoxes_[i][j].prob;
616             boundingBox.xmin = xmin;
617             boundingBox.ymin = ymin;
618             boundingBox.xmax = xmax;
619             boundingBox.ymax = ymax;
620             boundingBoxesResults_.bounding_boxes.push_back(boundingBox);
```

(a) Código Original

```
637             float xmin1 = (rosBoxes_[i][j].x - rosBoxes_[i][j].w / 2) * frameWidth_;
638             float ymin1 = (rosBoxes_[i][j].y - rosBoxes_[i][j].h / 2) * frameHeight_;
639             float xmax1 = (rosBoxes_[i][j].x + rosBoxes_[i][j].w / 2) * frameWidth_;
640             float ymax1 = (rosBoxes_[i][j].y + rosBoxes_[i][j].h / 2) * frameHeight_;
641
642             yolo_obstacle.type = classLabels_[i];
643             yolo_obstacle.probability = rosBoxes_[i][j].prob;
644             yolo_obstacle.x1 = xmin1;
645             yolo_obstacle.y1 = ymin1;
646             yolo_obstacle.x2 = xmax1;
647             yolo_obstacle.y2 = ymax1;
648             yolo_obstacle.h = 0.0;
649             yolo_obstacle.w = 0.0;
650             yolo_obstacle.l = 0.0;
651             yolo_obstacle.tx = 0.0;
652             yolo_obstacle.ty = 0.0;
653             yolo_obstacle.tz = 0.0;
654             yolo_listResults_.yolo_list.push_back(yolo_obstacle);
```

(b) Código añadido

Figura 3.19: Asignar valores a los campos del mensaje

Y por último, en el apartado donde se publica el topic (Figura 3.20a), se añade al final de estos, la publicación del topic creado (Figura 3.20b).

```
624     boundingBoxesResults_.header.stamp = ros::Time::now();
625     boundingBoxesResults_.header.frame_id = "detection";
626     boundingBoxesResults_.image_header = headerBuff_[(buffIndex_ + 1) % 3];
627     boundingBoxesPublisher_.publish(boundingBoxesResults_);
```

(a) Código Original

```
666     yolo_listResults_.header.stamp = ros::Time::now();
667     yolo_listResults_.header.frame_id = "detection";
668     yolo_listResults_.image_header = headerBuff_[(buffIndex_ + 1) % 3];
669     yoloListPublisher_.publish(yolo_listResults_);
```

(b) Código añadido

Figura 3.20: Publicar el topic

Con esto, el fichero fuente del paquete está modificado. Ahora hay que modificar el fichero cabecera que se encuentra en la ruta $ROS_WORKSPACE/src/darknet_ros/darknet_ros/include/darknet_ros/' y se llama '**YoloObjectDetector.hpp**'. Primero hay que incluir los mensajes nuevos (Figura 3.21b) debajo del bloque donde se incluyen el resto de mensajes (Figura 3.21a).

```
37  #include <darknet_ros_msgs/BoundingBoxes.h>
38  #include <darknet_ros_msgs/BoundingBox.h>
39  #include <darknet_ros_msgs/CheckForObjectsAction.h>
```

(a) Código Original

```
41  #include <darknet_ros_msgs/yolo_list.h>
42  #include <darknet_ros_msgs/yolo_obstacle.h>
```

(b) Código añadido

Figura 3.21: Declarar los mensajes en el include del paquete

En el bloque donde se instancian los publicadores y subscriptores (Figura 3.22a), se añade al final de este nuestro publicador (Figura 3.21b). Y en el bloque donde se declaran los objetos con las detecciones (Figura 3.23a), se añade nuestro objeto que contendra todas las detecciones con el tipo de mensaje que se creo (Figura 3.23b).

```
151     ros::Publisher objectPublisher_;
152     ros::Publisher boundingBoxesPublisher_;
```

(a) Código Original

```
158     ros::Publisher yoloListPublisher_;
```

(b) Código añadido

Figura 3.22: Declarar el publicador

```
155     std::vector<std::vector<RosBox_> > rosBoxes_;
156     std::vector<int> rosBoxCounter_;
157     darknet_ros_msgs::BoundingBoxes boundingBoxesResults_;
```

(a) Código Original

```
166     darknet_ros_msgs::yolo_list yolo_listResults_;
```

(b) Código añadido

Figura 3.23: Declaración del mensaje que contiene las detecciones de una iteracción

Ahora, hay que modificar el fichero de configuración '**ros.yaml**' que se localiza en la ruta $ROS_WORKSPACE/src/darknet_ros/darknet_ros/config/', donde hay que cambiar el topic al que se subscribira el nodo para obtener la imagen (Figura 3.24).

```
3   camera_reading:
4     topic: /camera/rgb/image_raw
5     queue_size: 1
```

(a) Código Original

```
3   camera_reading:
4     topic: /zed_node/left/image_rect_color
5     queue_size: 1
```

(b) Código modificado

Figura 3.24: Subscripción al topic de la ZED en el fichero de configuración del paquete Darknet

Finalmente, se compila el paquete para guardar todos los cambios.

3.6.2.4 Descargar los pesos de la YoloV3

Para hacer mas ligera la descarga del paquete de la Darknet, no incluye por defecto los pesos de ninguna red neuronal, por lo que hay que añadirlas. Se puede hacer de 2 maneras, descargarlas manualmente desde la página oficial [12] o desde el terminal y se deben guardar en la ruta '$ROS_WORKSPACE/src/darknet_ros/darknet_ros/yolo_network_config/weights/'. En el caso de hacerlo desde terminal, el código es el siguiente:

```
cd $ROS_WORKSPACE/src/darknet_ros/darknet_ros/yolo_network_config/weights/
wget http://pjreddie.com/media/files/yolov3.weights
```

3.7 Arquitectura de la comunicación

En la figura 3.25, se puede ver representado el esquema de comunicación entre los nodos que publica la ZED y la YOLO. El nodo **/zed_node** publicará una serie de topics con toda la información de la cámara y el nodo **/darknet_ros** publicará otra series de topics de la salida de la YOLO más el topic **/darknet_ros/yolo_list** [13] que es creado para publicar la información en un formato que el resto de los módulos del sistema global comparten.

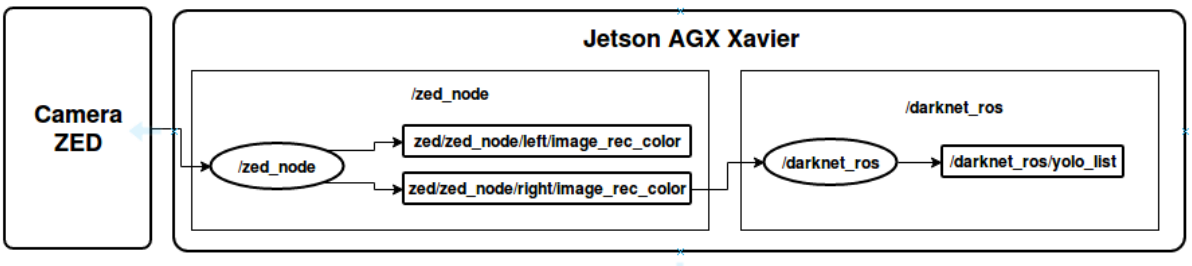


Figura 3.25: Esquema de nodos y topics publicados en la Jetson Xavier que usarán el resto de los módulos del sistema

[13] en la página 18 se explica como crear ese topic

Capítulo 4

Manual de Usuario

Para llegar a este capítulo primero hay que realizar el capítulo **??** (pág **??**). Este manual va dirigido a los miembros que vayan a comunicar la Jetson con los distintos módulos del proyecto y con los miembros que hayan desarrollado una nueva red neuronal con la arquitectura de Yolo y que hayan mejorado el set de entrenamiento para que detecte nuevas clases.

4.1 Modificar la velocidad de cómputo de la Jetson

La Jetson permite modificar el uso y velocidad de sus recursos en tiempo real, para así encontrar un compromiso entre potencia de computo y consumo. Por defecto, la Jetson se encuentra en el modo 2 con

NVPMODEL CLOCK CONFIGURATION

Mode Name	EDP	10W	15W	30W	30W	30W	30W
	MAXN	MODE_10W	MODE_15W	MODE_30W_ALL	MODE_30W_6CORE	MODE_30W_4CORE	MODE_30W_2CORE
Power Budget	n/a	10W	15W	30W	30W	30W	30W
Mode ID	0	1	2	3	4	5	6
Number of Online CPUs	8	2	4	8	6	4	2
CPU Maximal Frequency (MHz)	2265.6	1200	1200	1200	1450	1780	2100
GPU TPC	4	2	4	4	4	4	4
CPU Maximal Frequency (MHz)	1377	520	670	900	900	900	900
DLA Cores	2	2	2	2	2	2	2
DLA Maximal Frequency (MHz)	1395.2	550	750	1050	1050	1050	1050
Vision Accelerator (VA) cores	2	0	1	1	1	1	1
VA Maximal Frequency (MHz)	1088	0	550	760	760	760	760
Memory Maximal Frequency (MHz)	2133	1066	1333	1600	1600	1600	1600

Figura 4.1: Configuración de los recursos de la Jetson AGX Xavier

un consumo de 15W, como se puede apreciar en la figura 4.1. Pero como la Jetson irá embebido en un coche, no tendrá problemas de limitaciones de consumo, se configurará en el modo 0 para que vaya a la máxima potencia con el siguiente comando.

```
sudo nvpmodel -m [mode]
```

Donde [mode], será el modo que indica la segunda fila de la figura 4.1, que en este caso será:

```
sudo nvpmodel -m 0
```

Y para comprobar el modo actual se escribirá:

```
sudo nvpmodel -q
```

4.2 Lanzar el Sistema

4.2.1 Conexión Distribuida en ROS

Primero hay que configurar las ip de ros, porque en el apartado 3.4.1.2 (pág 15) se configuró las ip en localhost ya que inicialmente para hacer pruebas es suficiente, pero ahora hay que comunicar los topic publicados con los distintos módulos del proyecto.

Hay que asignar la ip con la que se publicarán los topics y será la ip que tenga la Jetson[1]. Para ello, hay que modificar el fichero **.bashrc**

```
nano ~./bashrc
```

Y sustituir <IP_JETSON>(Figura 4.2a) por la ip de la Jetson. Por último hay que asignar la ip que tenga el nodo maestro al que se conectarán el resto de nodos. Existe la posibilidad que ese nodo lo ejecute la Jetson o que lo ejecute otro sistema. Sea cual sea, hay que sustituir <IP_NODO_MAESTRO>(Figura 4.2b) por la ip donde se lance ese nodo.

```
export ROS_IP=<IP_JETSON>
```

(a) IP por la que la Jetson publica los topics

```
export ROS_MASTER_URI=http://<IP_NODO_MAESTRO>:11311
```

(b) IP donde se conectarán los nodos del sistema

Figura 4.2: Configuración para la conexión distribuida de ROS

Por último, para aplicar los cambios se ejecuta el siguiente comando:

```
source ~/.bashrc
```

4.2.2 Ejecutar el nodo de la ZED

Con la configuración de red realizada, solo queda lanzar el launch de la zed para que empiece a publicar por los topics lo que percive la cámara.

```
roslaunch zed_display_rviz display.launch
```

[1] Con el comando 'ipconfig' se puede consultar la ip

4.2.3 Ejecutar el nodo de la Darknet

Con los topics publicados por la ZED, se ejecuta la Darknet.

```
roslaunch darknet_ros yolo_v3.launch
```

Con esto, la Darknet empezará a publicar el topic **yolo_list** que contiene las detecciones obtenidas por la ZED y procesadas por la Darknet con el formato de mensaje creado.

4.3 Modificar la Darknet

La arquitectura de la Yolo y los pesos de la red instalados son de la version 3 con los pesos de la COCO, pero estos se pueden cambiar a otra arquitectura o a otros pesos, ya sean los que hay en la propia página de Yolo[12] o una nueva red entrenada. En ambos casos el procedimiento es el mismo.

4.3.1 Guardar la nueva red

Considerando que se esta localizado en la ruta donde se ubica el paquete de ros, Darknet, se guardarán los siguientes ficheros descargados de la página de Yolo u obtenidos de un entrenamiento:

- **ejemplo.names:** Contiene las clases que detecta la red. Se debe de ubicar en la carpeta 'darknet/data'
- **ejemplo.cfg:** Contiene la estructura de la red. Se debe ubicar en la carpeta 'darknet_ros/yolo_network_config/cfg'
- **ejemplo.weights:** Contiene los pesos de la red. Se debe ubicar en la carpeta 'darknet_ros/yolo_network_config/weights'

4.3.2 Ejecutar la nueva red

Con los ficheros anteriores, la nueva red puede ser utilizada, pero antes hay que crear un fichero de configuración para que el paquete Darknet pueda detectarla y un fichero launch para ejecutarlo facilmente.

Primero, hay que crear un fichero **ejemplo.yaml** en la ruta 'darknet_ros/config' con el formato de la figura 4.3

```
yolo_model:

  config_file:
    name: ejemplo.cfg
  weight_file:
    name: ejemplo.weights
  threshold:
    value: 0.3
  detection_classes:
    names:
      - clase1
      - clase2
      - clase3
```

Figura 4.3: Formato de configuracion de la nueva red de Yolo

Donde pone clase1, clase2, etc, hay que poner el nombre de las clases que contiene el fichero ejemplo.names.

Y por último, hay que crear el fichero **ejemplo.launch** en la ruta 'darknet_ros/launch' con el formato de la figura 4.4

```
<?xml version="1.0" encoding="utf-8"?>

<launch>

  <arg name="network_param_file"      default="$(find darknet_ros)/config/ejemplo.yaml"/>

  <include file="$(find darknet_ros)/launch/darknet_ros.launch">
    <arg name="network_param_file"    value="$(arg network_param_file)"/>
  </include>

</launch>
```

Figura 4.4: Formato de launcher para ejecutar la red

Y se compila todos los cambios

```
cd $ROS_WORKSPACE
catkin_make
rospack_profile
```

Ahora, para ejecutar la nueva red, el nodo de la ZED debe de estar ejecutandose y se debe escribir el siguiente comando:

```
roslaunch darknet_ros ejemplo.launch
```

Capítulo 5

Resultados

Para la fase de testeo del sistema, primero se realizó una grabación de 30s de uno de los topic que publicaba la ZED con el comando:

```
rosbag record /zed/zed_node/left/image_rec_color zed.bag --duration=30
```

Con esto se pretende realizar pruebas lo más objetivamente posible. La idea es ejecutar distintas arquitecturas de la YOLO para comprobar su calidad de detección y velocidad de procesado en la Jetson Xavier

Se utilizaron la YOLOv2 y la YOLOv3, donde esta última es un poco más lenta, pero detecta con más fiabilidad, y además, se usaron la versión 'tiny' de ambas, que son capaces de detectar las mismas clases respectivamente, pero son más rápidas al tener menos capas en su arquitectura y esto hace que pierdan calidad de detección. Aun así, dependiendo con que dataset se hayan entrenado, una version 'tiny' puede ser una buena opción.

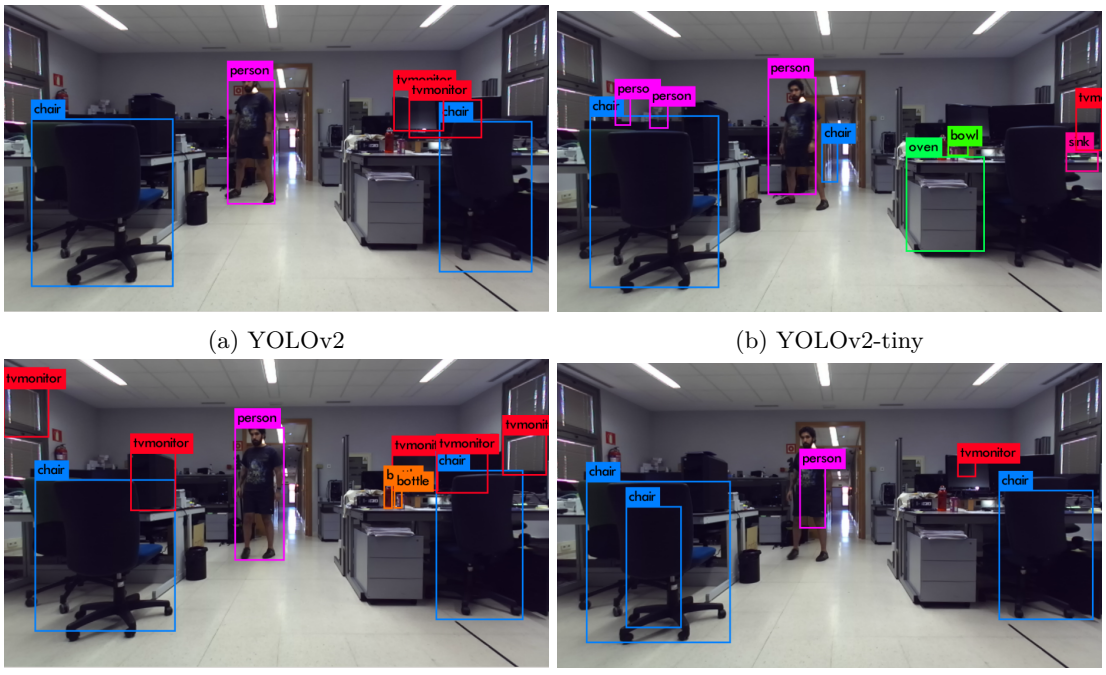


Figura 5.1: Comparación entre las distintas arquitecturas de la YOLO

Las imagenes de la Figura 5.1 fueron ejecutadas en la Jetson Xavier y de esta prueba se saca varias conclusiones en la calidad de detección:

- La versiones 'tiny' tienen fallos de deteccion y fallos en las dimensiones de la bounding box
- La YOLOv2 es la que mejor acierto tiene de detecciones y del tamaño de la bounding box
- la YOLOv3 detecta objetos que ninguna otra red detectaba, como en el caso de las botellas, que efectivamente habia varias botellas en la mesa, aunque tambien se equivoca en las detecciones de tvmonitor, ya que confunde a varias con las ventanas.

Luego se hizo una prueba de velocidad de procesado. Se realizó la misma prueba para las cuatro arquitecturas anteriores de la YOLO y aprovechando que en el laboratorio estaba tanto la Jetson Xavier como la Jetson TX2 con la misma instalación y configuración, se hizo la misma prueba en ambas con todos los modos de configuración de recursos.

Jetson Xavier: Se utilizó información de la figura 4.1 como referencia y estos fueron los resultados.

Jetson AGX Xavier							
CNN	MODO						
	0	1	2	3	4	5	6
YOLOv2	16 fps	3,2 fps	8 fps	11 fps	11,2 fps	11,3 fps	9,7 fps
YOLOv2-tiny	50 fps	10 fps	20 fps	22 fps	33 fps	32 fps	16 fps
YOLOv3	7,7 fps	1,8 fps	4 fps	5,4 fps	5,4 fps	5,4 fps	5,2 fps
YOLOv3-tiny	43 fps	8,5 fps	20 fps	22 fps	31 fps	31 fps	16 fps

Tabla 5.1: Velocidad de procesado de la Jetson AGX Xavier

Jetson TX2 Se utilizó información de la figura 5.2 como referencia y estos fueron los resultados.

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

Figura 5.2: Configuración de los recursos de la Jetson TX2

Jetson TX2					
CNN	MODO				
	0	1	2	3	4
YOLOv2	7,6 fps	5 fps	6,5fps	6,5 fps	6,5 fps
YOLOv2-tiny	19,8 fps	13,4 fps	13,2 fps	17 fps	18,9 fps
YOLOv3	3,3 fps	2,2 fps	2,9 fps	2,9 fps	2,9 fps
YOLOv3-tiny	20 fps	13,2 fps	13,1 fps	17 fps	18,7 fps

Tabla 5.2: Velocidad de procesado de la Jetson TX2

Como curiosidad, en la Jetson TX1, su modo 0 equivaldría al modo 1 de la Jetson TX2. Esto se debe a varios factores y no simplemente a la velocidad de reloj, por ejemplo, el ancho de banda del bus de memoria es de 128-bits en la Jetson TX2 y de 64-bits en la Jetson TX1.

Finalmente, la Jetson se montó en el coche autónomo y se conecto con el resto de módulos del sistema. La versión usada fue la YOLOv2 para asegurarse de que no procesara las imágenes lentamente y así poder probar todos los módulos del que se compone el vehículo.

(a) Resultado de la YOLO 1

(b) Resultado de la YOLO 2

Figura 5.3: Resultado de la YOLOv2 con la Jetson montada en el vehículo

Capítulo 6

Conclusiones y Líneas futuras

6.1 Conclusiones

LA YOLO tiene un buen indice de acierto en las detecciones como se ha podido ver en la figura 5.1. En esa figura no aparece la probabilidad de acierto en la detección. Si por ejemplo, se umbralizará las detecciones con un acierto de un 70 % o superior, desecharía todas esa detecciones que se ven que están mal, como en el caso de la YOLOv3 en la figura 5.1c.

También se ha intentado aumentar el número de fps de la YOLOv3 sobre la Xavier, reduciendo el número de clases a detectar. Esta prueba se ha realizado usando el video de testeo del capítulo anterior, haciendo que solo detecte la clase 'person', pero no ha variado prácticamente nada.

En los resultados de procesado de la Jetson Xavier, se puede ver perfectamente como es más rápida que la Jetson TX2, pero esto no es ninguna novedad ya que en las especificaciones técnicas se puede contrastar la diferencia entre ambas (página 37), pero centrandonos únicamente en la Jetson Xavier se han encontrado pros y contras frente al portatil MSI donde anteriormente se ejecutaba:

Pros

- Menor Tamaño. Tiene unas dimensiones bastante reducidas, por lo que se puede colocar en cualquier parte del coche sin que moleste.
- Menor consumo. Si se considera la peor situación, el mayor consumo se obtendría si se configura la Jetson Xavier en el modo 0, y aunque en la figura 4.1 no especifica el consumo en ese modo, se puede asumir que el consumo es inferior a los 60W que es capaz de suministrar su fuente de alimentación, ya que la Jetson también tiene que ser capaz de alimentar los periféricos que tenga conectados. Esto comparado con los 300W que puede suministrar el portatil MSI.

Contras

- Velocidad de procesado. El portail MSI procesa a 20 fps la YOLOv3 frente a los 7,7 fps que procesa Jetson Xavier en el modo 0. No es una velocidad perfecta, pero si se puede emplear en el vehículo autónomo.

6.2 Líneas futuras

Una posible forma de hacer que mejore la YOLOv3 es entrenandola con un dataset específico del objeto u objetos que se quieren detectar, e incluso en su versión 'tiny' puede que tenga buenos resultados de

detección, porque de velocidad de proceso va sobrado.

Y un buen punto a mejorar, sería instalar todo el sistema en un **docker**, por diversos motivos.

1. La difusión del sistema montado y configurado con otras placas, ya sea entre el mismo modelo u otras que compartan un hardware similar.
2. La gestión de versiones.

Solamente, habría que haber flasheado la Jetson y en la parte software, haber instaldo docker. Con esto habría sido suficiente y, a partir de ahi, trabajar desde el docker. Si la versión actual del sistema funciona y es estable, se hace un 'commit' y se guarda la versión, que ha salido algo mal, se vuelve a una versión pasada donde el sistema era estable sin tener que volver a empezar de cero. Justamente esta última parte me habría ahorrado flashear 3 veces la Jetson.

El problema que había era, que la versión actual del docker para la Jetson no permitía la computación con la GPU, y justamente este es uno de los atractivos que tiene el usar la Jetson, así que hubo que instalar todo de forma nativa.

Bibliografía

[1] Carlos Pérez de Rivas, 2018, *"Detección y tracking de objetos utilizando visión y láser 3D para vehículos inteligentes"* (Trabajo Fin de Grado), Universidad de Alcalá, Alcalá de Henares.

[2] The KITTI Vision Benchmark Suite,
http://www.cvlibs.net/datasets/kitti/index.php

[3] Grupo Robesafe,
https://www.robesafe.uah.es/index.php/en/

[4] Rafael Barea, Luis M. Bergasa, Elena López-Guillén, Eduardo Romera, Joaquín López, *"Smart Elderly Car: The UAH Intelligent Electric Vehicle"*,
https://www.robesafe.uah.es/index.php/en/smartelderlycar

[5] NVIDIA Jetson Xavier,
https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit

[6] Instalador del JetPack,
https://developer.nvidia.com/embedded/jetpack

[7] Configuración recursos Jetson AGX Xavier,
https://www.jetsonhacks.com/2018/10/07/nvpmodel-nvidia-jetson-agx-xavier\
-developer-kit/

[8] ROS Melodic Morenia,
http://wiki.ros.org/melodic

[9] Stereolabs, ZED Stereo Camera,
https://www.stereolabs.com/

[10] ZED SDK,
https://www.stereolabs.com/developers/release/

[11] Stereolabs ZED Camera - ROS Integration,
https://wiki.ros.org/zed-ros-wrapper

[12] YOLOv3: An Incremental Improvement, Redmon, Joseph and Farhadi, Ali, 2018
https://pjreddie.com/darknet/yolo/

[13] YOLO ROS: Real-Time Object Deteccion for ROS, Marko Bjelonic, 2016/2018
https://github.com/leggedrobotics/darknet_ros

Apéndice A

Herramientas y recursos

Las herramientas necesarias para la elaboración del proyecto han sido:

- PC Host con sistema operativo GNU/Linux 16.04
- NVIDIA Jetson Xavier
- ZED Camera. Cámara RGB estereoscópica
- YOLO. Real Time Object Detection: Red Neuronal Convolucional
- ROS: Framework para el desarrollo del software
- Visual Studio Code: Editor de código
- Texmaker: Procesador de textos LaTeX

Apéndice B

Especificaciones Técnicas

Especificaciones Técnicas Jetson Nano	
GPU	Arquitectura NVIDIA Maxwell con 128 núcleos NVIDIA CUDA
CPU	Procesador ARM® Cortex®-A57 MPCore de cuatro núcleos
Memoria	LPDDR4 de 4 GB y 64 bits
Almacenamiento	16 GB de almacenamiento Flash eMMC 5.1
Codificación de vídeo	4K a 30 cuadros (H.264/H.265)
Descodificación de vídeo	4K a 60 cuadros (H.264/H.265)
Cámara	12 vías (3 x 4 o 4 x 2) MIPI CSI-2 DPHY 1.1 (18 Gbps)
Conectividad	Gigabit Ethernet
Pantalla	HDMI 2.0 o DP 1.2 | eDP 1.4 | DSI (1 x 2) 2 simultáneos
UPHY	1 1/2/4 PCIE, 1 USB 3.0, 3 USB 2.0
E/S	1 SDIO / 2 SPI / 4 I2C / 2 I2S / GPIO
Tamaño	69,6 mm x 45 mm
Mecánicas	Conector de 260 pines

Tabla B.1: Especificaciones técnicas Jetson Nano

Especificaciones Técnicas Serie Jetson TX2			
	TX2(4GB)	**TX2**	**TX2i**
GPU	Arquitectura NVIDIA Pascal con 256 núcleos NVIDIA CUDA		
CPU	CPU de 64 bits Denver 2 de doble núcleo y ARM A57 Complex		
Memoria	LPDDR4 de 4 GB y 128 bits	LPDDR4 de 8 GB y 128 bits	
Almacenamiento	eMMC 5.1 de 16 GB	eMMC 5.1 de 32 GB	
Cod. de vídeo	3 de 4K a 30 cuadros (HEVC)		
Descod. de vídeo	4x 4K @ 30 (HEVC)		
Conectividad	Wi-Fi no integrado	Wi-Fi integrado	Wi-Fi no integrado
	Ethernet		
Cámara	12 vías MIPI CSI-2, D-PHY 1.2 (30 Gbps)		
Pantalla	HDMI 2.0/eDP 1.4/2 DSI/2 DP 1.2		
UPHY	Gen 2 | 1x4 + 1x1 O 2x1 + 1x2, USB 3.0 + USB 2.0		
Tamaño	87 mm x 50 mm		
Mecánicas	Conector de 400 pines con placa de transferencia térmica (TTP)		

Tabla B.2: Especificaciones técnicas Jetson TX2

Especificaciones Técnicas Serie Jetson Xavier		
	Jetson AGX Xavier 8GB	**Jetson AGX Xavier**
GPU	384-Core Volta GPU with 48 Tensor cores 5.5 TFLOPS (FP16) 11.1 TOPS (INT8)	512-Core Volta GPU with 64 Tensor Cores 11 TFLOPS (FP16) 22 TOPS (INT8)
Acelerador DL	(2x) NVDLA Engines 4.1 TFLOPS (FP16) 8.2 TOPS (INT8)	(2x) NVDLA Engines 5 TFLOPS (FP16) 10 TOPS (INT8)
CPU	6-core ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3	8-Core ARM v8.2 64-Bit CPU, 8MB L2 + 4MB L3
Memoria	8GB 256-bit LPDDR4x, 1333MHz - 85GB/s	16GB 256-bit LPDDR4x, 2133MHz - 137GB/s
Pantalla	Three multi-mode DP 1.2/eDP 1.4/HDMI 2.0	
Almacenamiento	32GB eMMC 5.1	
Acelerador de visión	7-Way VLIW Vision Processor	
Codificación de vídeo	2x 4K @ 30 (HEVC)	8x 4K @ 30 (HEVC)
Descodificación de vídeo	4x 4K @ 30 (HEVC)	12x 4K @ 30 (HEVC)
Cámara	16 lanes MIPI CSI-2, 8 lanes SLVS-EC D-PHY (40 Gbps) C-PHY (64 Gbps)	16 lanes MIPI CSI-2, 8 lanes SLVS-EC D-PHY (40 Gbps) C-PHY (109 Gbps)
UPHY	3xUSB 3.1, 4xUSB 2.0 1x8 or 1x4 or 1x2 or 2x1 PCIe (Gen3)	3xUSB 3.1, 4xUSB 2.0 1x8 or 1x4 or 1x2 or 2x1 PCIe (Gen4)
Otros	UART, SPI, CAN, I2C, I2S, DMIC, GPIOs	
Conectividad	10/100/1000 RGMII	
Tamaño	100 mm x 87 mm	
Mecánicas	Conector Molex Mirror Mex de 699 pines Placa de transferencia térmica integrada	

Tabla B.3: Especificaciones técnicas Jetson Xavier

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá