UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE
INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

DESARROLLO, IMPLEMENTACIÓN Y TESTEO DE UN SISTEMA DE DETECCIÓN Y LOCALIZACIÓN DE CONOS PARA UN VEHÍCULO FORMULA STUDENT

MÁSTER EN INGENIERÍA DE TELECOMUNICACIÓN

CARLOS PÉREZ MUÑOZ
MÁLAGA, 2020

Yo, Carlos Pérez Muñoz, estudiante del Máster en Ingeniería de Telecomunicación, de la Escuela Técnica Superior de Ingeniería de Telecomunicación de la Universidad de Málaga, declaro que no se ha utilizado contenido de fuentes, sin citarlas adecuadamente, en la realización de la presente memoria y, en consecuencia, la ausencia de plagio.

En Málaga a 22 de septiembre de 2020.

Fdo: Carlos Pérez Muñoz

E.T.S. DE INGENIERÍA DE TELECOMUNICACIÓN, UNIVERSIDAD DE MÁLAGA

Desarrollo, implementación y testeo de un sistema de detección y localización de conos para un vehículo Formula Student

Autor: Carlos Pérez Muñoz

Tutor: Martín González García

Departamento: Tecnología Electrónica

Titulación: Máster en Ingeniería de Telecomunicación

Palabras clave: Visión Artificial, CUDA, CNN, DNN, openCV

Resumen

Se busca desarrollar un subsistema que permita, a partir de una entrada de video, detectar un conjunto de conos ya conocidos, reconocer el tipo de cono y posicionarlos partiendo de unas dimensiones conocidas. La idea es integrar dicho subsistema en un sistema de conducción autónoma enfocado a la competición *Formula Student*.

Se persigue que el sistema ofrezca una precisión suficiente para trazar un mapa detallado del circuito, que funcione en tiempo real y que tenga el menor coste posible, tanto en términos de recursos como económicos.

Se ha desarrollado un programa basado en la API *openCV* que, a partir del fotograma obtenido de un archivo de imagen, video o entrada de cámara, muestra en tiempo real por pantalla dicha entrada resaltando los conos detectados con la información requerida. Dicho programa funciona en un PC con Linux y GPU dedicada Nvidia. Los resultados obtenidos, en términos de precisión de la detección y localización y de rendimiento en fotogramas por segundo, son satisfactorios teniendo en cuenta el hardware utilizado.

E.T.S. DE INGENIERÍA DE TELECOMUNICACIÓN, UNIVERSIDAD DE MÁLAGA

Development, implementation and testing of a cone detection and location system for a Formula Student vehicle

Author: Carlos Pérez Muñoz

Supervisor: Martín González García

Department: Electronic Technology

Degree: Master's degree in Telecommunications Engineering

Keywords: Computer Vision, CUDA, CNN, DNN, openCV

Abstract

We aim to develop a subsystem which allows, from a video entry, detect a set of well-known cones, their type and position using known measures. The idea is to integrate said subsystem within an autonomous driving system focused on the Formula Student competition.

We desire a system with enough accuracy to allow a detailed mapping of the circuit, which works in real time and with the lowest cost possible in terms of resources and money.

An OpenCV-based program has been developed which, from a photogram of an image file, video file or camera input, shows said input on screen in real time, highlighting detected cones with the required information. This program runs on a PC using Linux and a dedicated Nvidia GPU. The obtained results, in terms of detection and positioning accuracy and performance measured in frames per second, are satisfactory considering the hardware we used.

Agradecimientos

A mi familia, especialmente a mis padres, a mis compañeros de carrera y a mis amigos por el apoyo recibido. Al equipo de Málaga Racing Team y a mi tutor por ayudarme a idear y concretar los objetivos de este proyecto y ofrecerme apoyo y recursos durante su realización.

A mis padres, por haberme ayudado

Y apoyado todos estos años...

Contenido

Lista de Acrónimos

ANOVA	Analisys of Variance
API	Application Programming Interface
CNN	Convolutional Neural Network
CUDA	Compute Unified Device Architecture
DNN	Deep Neural Network
DPU	Deep Learning Processing Unit
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
GPU	Graphics Processing Unit
IA	Inteligencia Artificial
NMS	Non Maximum Suppression
PC	Personal Computer
YOLO	You Only Look Once

Lista de Figuras

Lista de Tablas

Capítulo 1. Introducción

1.1. Contexto del proyecto

El origen de este proyecto se puede atribuir al campo de la conducción autónoma. Este campo ha ganado mucha popularidad en los últimos años gracias los avances del hardware en cuanto a potencia, consumo, disponibilidad y precio; los avances hechos en inteligencia artificial y el interés de grandes compañías tecnológicas y automovilísticas, las cuales llevan unos años haciendo pruebas usando vehículos autónomos prototipo con resultados que prometen traerlo al público a lo largo de esta década. Este campo está tan avanzado que se considera que, para tareas de conducción sencillas, un equipo de estudiantes de ingeniería con fondos suficientes puede desarrollar un vehículo autónomo funcional.

Aquí entra la competición *Formula Student,* organizada a nivel nacional por varios países europeos. Se trata de una competición en la que equipos de estudiantes de cada universidad diseñan y fabrican un vehículo de carreras de acuerdo a una normativa preestablecida junto con un plan de negocio y compiten entre sí por ver qué equipo tiene el coche más rápido, el mejor diseñado y el mejor plan de negocio y financiación. La competición tiene varias categorías, que consisten en vehículos de combustión (CV), vehículos eléctricos (EV) y vehículos sin conductor (DV). Es esta última la que nos interesa, ya que el proyecto está orientado a formar parte de un vehículo autónomo diseñado para dicha competición.

La tarea de la conducción autónoma se puede dividir en varias subtareas tales como: establecimiento de una ruta y reglas a seguir, evaluación del entorno del vehículo, toma de decisiones de conducción con los datos disponibles y aplicar dichas decisiones al comportamiento del vehículo (aceleración, frenada, control del volante y cambio de marchas). Este proyecto se va a centrar en la subtarea de evaluación del entorno, concretamente en una pista de carreras Formula Student.

Si se echa un vistazo al reglamento y al manual de competición [1], éste explica que, en las pruebas en pista, dicha pista está delimitada por conos con modelos y funciones definidos. Esto implica que su detección, clasificación y posición nos proporcionará toda la información que necesitamos sobre la pista.

Por otra parte, otro campo que ha ganado mucha popularidad y desarrollo durante los últimos años ha sido el campo de la Inteligencia Artificial, más concretamente sus variantes *Machine Learning* y *Deep Learning*. Estas herramientas prometen resolver problemas que antes eran no eran factibles y los avances en el hardware y los algoritmos por fin permiten su uso generalizado en tiempo real para varias aplicaciones. Una de dichas aplicaciones es el reconocimiento de objetos, el cual va a ser un pilar fundamental de nuestro proyecto puesto que, para posicionar y clasificar los conos, primero hay que detectarlos.

1.2. Objetivos del proyecto

El objetivo de este proyecto es desarrollar un sistema que permita el reconocimiento y el posicionamiento de un conjunto concreto de conos a partir de una entrada de video con el fin ayudar a la conducción autónoma de un vehículo *Formula Student*. Entre las tareas se encuentran la detección de los conos en un fotograma (posición y bordes), su clasificación en uno de los cuatro tipos existentes y estimación de su posición en el plano del suelo respecto a la cámara a partir de los bordes del cono detectado y las dimensiones de éste ya conocidas.

Se pretende escribir un programa demo que muestre en tiempo real la entrada de un fichero de imagen o video o una entrada de video y, sobre dicha imagen,

muestre toda la información respecto a los conos encontrados. Dicho programa también tendrá un modo de calibración con el que, ejecutándolo una sola vez, nos permita trabajar con cualquier cámara independientemente de su óptica.

Con el programa funcional, se quiere realizar una serie de experimentos para caracterizar, y a ser posible mejorar, la precisión de las posiciones obtenidas, el alcance de la detección y el rendimiento del programa en fotogramas procesados por segundo. Por último, en base a los datos de rendimiento, se quiere estimar unos requisitos de hardware para cumplir el requisito de funcionamiento en tiempo real, que se ha establecido en 30 FPS (*Frames per Second*) constantes.

1.3. Estructura de la memoria

En primer lugar, se hará una introducción a las tecnologías, API (*Application Programming Interface*), modelos y reglas usados en el desarrollo del sistema. Después, se expondrá detalladamente el desarrollo del sistema diseñado y la demo que lo incluye. Posteriormente, se especificarán una serie de pruebas y se mostrarán los resultados y conclusiones obtenidos. A continuación, se redactarán una serie de guías y consejos de cara a la integración del proyecto en un programa de control de un vehículo autónomo. Por último, se expondrán las conclusiones y posibles mejoras.

Capítulo 2. Conceptos previos

En este capítulo se abordarán temas esenciales para la comprensión del proyecto y de las decisiones tomadas.

2.1. Formula Student

Puesto que nuestro sistema va a estar enfocado a la competición *Formula Student*, conviene explicar las reglas que van a impactar en las necesidades y limitaciones del proyecto. Como se expuso en el contexto, la competición usa un conjunto concreto de conos, cada uno con una función concreta. Estos conos son:

- Cono azul (WEMAS 400.000043.00.00): señala el lado izquierdo de la pista. Se muestra un ejemplo en la figura 2.1.
- Cono amarillo (WEMAS 400.000013.01.10): designa el lado derecho de la pista. Se muestra un ejemplo en la figura 2.2.
- Cono naranja chico (WEMAS 400.000013.00.00): delimitan las zonas de entrada y salida de pista. Se muestra un ejemplo en la figura 2.3.
- Cono naranja grande (WEMAS 307.610500.00.00): marcan las líneas de salida, meta y puntos de control para medida de tiempos. Se muestra un ejemplo en la figura 2.4.

Figura 2.1. Cono azul

Figura 2.2. Cono amarillo

Figura 2.3. Cono naranja chico

Figura 2.4. Cono naranja grande

Todos los conos chicos tienen unas dimensiones de $228mm \times 228mm \times 325mm$, mientras que el grande tiene unas dimensiones de $285mm \times 285mm \times 505mm$. En cuanto a su colocación, el reglamento establece un ancho de pista mínimo de tres metros y una distancia máxima entre conos consecutivos de cinco metros. Por último, puede que se encuentren conos apilados a un lado de la pista. Estos no presentan ninguna función. En la figura 2.5 se ilustra mejor la colocación descrita.

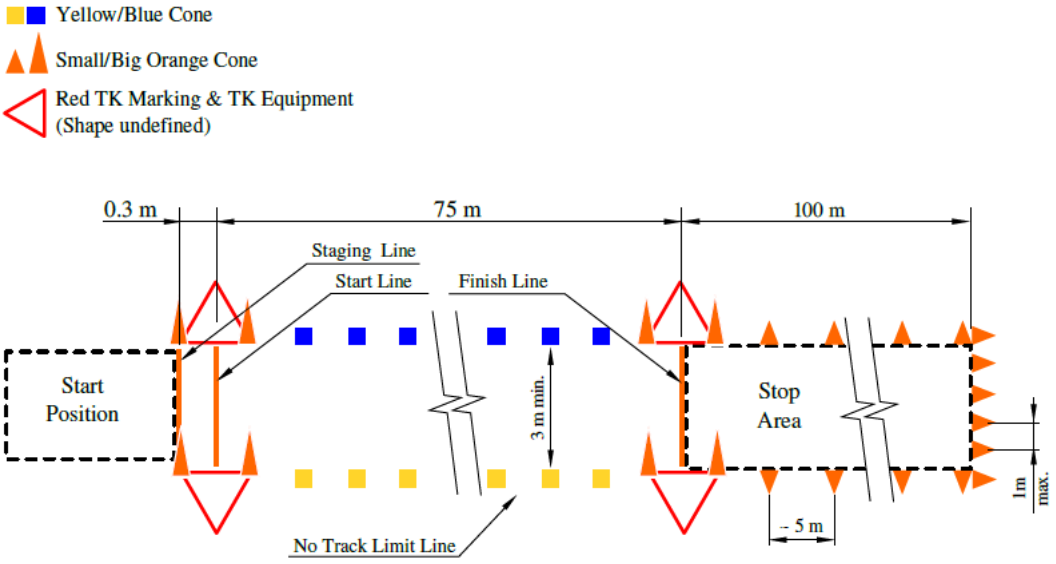

Figura 2.5. Ejemplo de disposición de conos (Fuente: [1])

2.2. La API *openCV*

El problema fundamental del proyecto entra dentro del campo conocido como visión artificial, esto es, la ciencia y la técnica de extraer información relevante de una imagen o video de forma automática por medio de un sistema informático.

OpenCV es la API de visión artificial más popular y utilizada. Es orientada a objetos, de código abierto, multiplataforma, multilenguaje (soporta C++, Java y Python) y cuenta con una gran comunidad, documentación exhaustiva y numerosos tutoriales. Entre las funcionalidades que ofrece, nos interesan las siguientes:

- Representación de las imágenes y fotogramas mediante un objeto matriz multidimensional que abstrae toda la gestión de memoria del usuario.
- Fácil lectura de ficheros de imagen o video y entradas de video presentes en el sistema.
- Igualmente, permite escribir imágenes o videos en ficheros y mostrarlos por pantalla.
- Utilidades de dibujo para presentar información en la propia imagen.
- Gran optimización del procesado de imagen y video sobre CPU.
- Gran soporte y facilidad de uso de redes neuronales, sin importar su *framework* o entorno de ejecución, incluyendo soporte para Nvidia CUDA y openCL para aceleración por GPU.

Todas estas funcionalidades alivian considerablemente la carga de trabajo en el desarrollo del proyecto.

2.3. Redes neuronales convolucionales

El problema más complejo a resolver en este proyecto es también el más inmediato: la detección de conos en una imagen. Tratándose de un problema de detección de objetos, una de las soluciones más populares y efectivas es el uso de redes neuronales convolucionales o CNN (Convolutional Neural Network). Aunque su funcionamiento en detalle queda fuera del alcance de este trabajo, conviene conocer en qué consisten, aunque sea de forma superficial.

Las redes neuronales son la herramienta utilizada en el campo del *Deep Learning*, que es un subcampo del *Machine Learning,* que es a su vez otro subcampo de la IA (Inteligencia Artificial) o AI (*Artificial Intelligence)* en inglés [2]. La IA consiste, en su definición más básica, en la toma automática de decisiones por parte de un computador en base a unos datos de entrada y a un conjunto de reglas que rige dicha toma de decisiones. Este campo existe desde los años 50. La aproximación clásica al problema de la IA ha sido la codificación manual de las reglas de decisión en un programa informático. El problema surge cuando las reglas para resolver el problema en cuestión son demasiado complejas o no están claras. Con la intención de resolver este obstáculo, surgió el ML (*Machine Learning*) en los años 90.

El *Machine Learning* consiste en una nueva aproximación a la IA que busca resolver el problema de las reglas. En lugar de codificar manualmente dichas reglas, la IA es “entrenada” a partir de unos datos de entrada y unas salidas esperadas con los que infiere una serie de reglas que produzcan un resultado lo más parecido posible. Para ello, hace uso, aparte de los datos de entrada y salida, de una métrica de error que cuantifica la diferencia entre la salida esperada y la obtenida con el conjunto de reglas actual. El sistema va cambiando las reglas en sucesivas iteraciones con la esperanza de encontrar aquellas que reduzcan la métrica del error. Cuando la IA es desplegada para su uso, lo que se conoce como inferencia en contraposición al entrenamiento, lleva a cabo operaciones y transformaciones sobre los datos de entrada hasta obtener una representación que permita una mejor toma de decisión.

El *Deep Learning* es, en realidad, un modelo de *Machine Learning* más concreto. En él, las operaciones sobre los datos de entrada se realizan de forma sucesiva y se encuentran organizadas en “capas”. Cuando los datos son procesados por una capa, el resultado de dicha capa es más representativo de cara a la inferencia que el resultado de la capa anterior. Los algoritmos de *Deep Learning* se suelen organizar en redes neuronales, que consisten en el apilamiento de dichas capas. A pesar del nombre, no existen pruebas de que el cerebro humano tenga una organización similar. El número de capas que componen la red se denomina “profundidad” de la red (*Depth* en inglés, de ahí viene el nombre de *Deep Learning*). En Deep *Learning*, las reglas se componen de dos elementos: las operaciones que llevan a cabo cada capa y los “pesos” o parámetros que usa para llevar a cabo dichas operaciones. En la fase de entrenamiento, con cada iteración se ajustan los pesos. En otras palabras, el aprendizaje de una red neuronal consiste en encontrar los pesos de las capas que ofrezcan el resultado más parecido al deseado.

Por último, las redes neuronales convolucionales son un tipo especial de red neuronal cuya aplicación principal se da en el campo de la visión artificial para resolver problemas de clasificación de imágenes y detección de objetos en una imagen. Se llaman así porque estas redes utilizan principalmente la operación convolución que, en procesado de imagen, es calcular cada píxel como la suma ponderada de sí mismo y los píxeles que lo rodean. Estas redes son la herramienta elegida para resolver el problema de la detección y clasificación de conos.

2.3.1. YOLO y YOLO-tiny

Una vez que se ha decidido usar una CNN, hay que decidir cuál usar para después entrenarla y desplegarla como parte de nuestro sistema. Buscamos una red que ofrezca 2 características:

- Alta precisión: Entendemos por precisión la capacidad de detectar objetos sin falsos positivos ni falsos negativos, así como que su delimitación se ajuste a los bordes reales del objeto. Esto último es crucial para un buen posicionamiento, ya que se va a basar en las dimensiones de los conos.

- Alto rendimiento: es deseable que la red sea capaz de funcionar en tiempo real (más de 30 imágenes analizadas por segundo) usando la menor cantidad de recursos posible, con el fin de que el sistema final sea lo más económico posible.

Cuando se busca una red neuronal para reconocimiento de objetos en tiempo real, la alternativa más popular en los últimos tres años han sido las redes YOLO. Sus creadores presumían de ofrecer una precisión similar a otras redes ya existentes, pero siendo capaz de funcionar en tiempo real haciendo uso de una Unidad de Procesado Gráfico (GPU) de consumidor. YOLO viene a significar *You Only Look Once* (solo miras una vez). El nombre viene de que, a diferencia de las redes existentes en el momento de su publicación, esta red solo evalúa la imagen una vez, contribuyendo enormemente al aumento de eficiencia. Desde la publicación de la primera versión de la red en 2016, han aparecido cuatro nuevas versiones, cada una optimizando la precisión y el rendimiento de la red anterior añadiendo nuevas técnicas y operaciones que han mostrado ser eficiente. En concreto, la cuarta versión (la más reciente) asegura ser un 10% más precisa y un 12% más rápida que la versión anterior [3].

El único inconveniente de esta red es que sigue requiriendo una GPU de gama media-alta para funcionar en tiempo real. Además, estas redes se diseñan con objetos complejos y muy variables en mente, como caras, vehículos y animales. En nuestro caso, los conos son siempre los mismos y su apariencia apenas cambia con la posición. Esto quiere decir que nos conviene más una red más simple. Por suerte, los autores de YOLO publican, junto con cada versión de la red, una versión pequeña o *tiny*, que permite su ejecución en sistemas más limitados a costa de una peor precisión.

Al final, la red elegida ha sido YOLOv4-tiny. Como se verá más adelante, ofrece la precisión suficiente y es considerablemente eficiente en cuanto a recursos. No se han considerado otras redes debido a que sus requisitos de potencia son iguales o superiores a los de la red YOLO estándar.

2.4. GPU y Nvidia CUDA

Para la tarea de detección de objetos en tiempo real, tener una red neuronal precisa y eficiente no es suficiente. Hace falta un hardware suficientemente potente sobre el que ejecutarse, tanto para el entrenamiento como la inferencia. Las redes neuronales despegaron hace diez años por la disponibilidad de hardware con la potencia necesaria. El componente clave fueron las GPU programables para propósito general.

Una GPU es, en términos sencillos, un procesador orientado a tareas de cómputo de imágenes y escenas 3D. Las primeras GPU se comercializaron en 1999 de la mano de Nvidia. En aquella época, las GPU eran diseñadas para un propósito concreto, los videojuegos, y funcionaban de forma inflexible.

Todo cambió en 2007 con el lanzamiento de la arquitectura Tesla y Nvidia CUDA. Esta arquitectura presentaba un nuevo paradigma de diseño. En vez de una arquitectura rígida como se usaba hasta entonces, la arquitectura Tesla consiste principalmente en varios ciento de núcleos sencillos programables capaces de trabajar en paralelo. CUDA (*Compute Unified Device Architecture*) es una API basada en C/C++ que permite a desarrolladores usar las enormes capacidades de computación en paralelo de las nuevas GPU Nvidia. Esta nueva herramienta no solo es de utilidad para desarrolladores de videojuegos, sino que rápidamente se abrió paso en la comunidad científica por su utilidad, entre otros campos, de potenciar simulaciones físicas, hasta tal punto que software como Matlab lo tienen incorporado. En nuestro caso, el uso de CUDA como *backend* de ejecución en openCV mejora enormemente el rendimiento y en el caso del software de entrenamiento de YOLO es prácticamente un requisito.

2.5. Modelo de cámara estenopeica

Una vez resuelto el problema de la detección y clasificación de los conos, queda el problema de estimar su posición. Para ello, hace falta un modelo matemático que relacione posición de los conos en la imagen 2D con su imagen en el mundo real 3D. Un modelo sencillo pero suficiente es el modelo de cámara estenopeica (*pinhole camera* en inglés).

Este modelo [4] considera que una cámara consiste en un plano de origen opaco con un punto infinitesimalmente pequeño por el que pasa la luz, llamado apertura de la cámara, y un plano de imagen detrás del plano de origen donde se proyecta la imagen. En este modelo, la imagen de un objeto percibida por la cámara consiste en su proyección en el plano de imagen a través de la apertura, como se muestra en la figura 2.5.

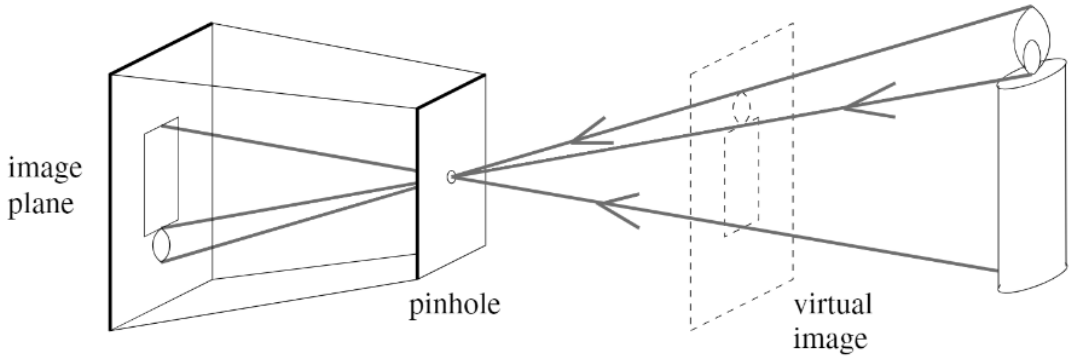

Figura 2.6. Modelo de cámara estenopeica general (Fuente: [3])

Como se observa en la figura 2.5, la imagen proyectada está invertida con respecto al objeto real. Para corregirlo dentro del modelo, se coloca el plano de imagen delante de la apertura a la misma distancia que antes. A la hora de realizar cálculos con el modelo, se hacen las siguientes suposiciones:

- La apertura focal está en el origen de coordenadas.
- El sistema de coordenadas consiste en tres coordenadas (*x1*, *x2* y *x3*) donde *x1* y *x2* forman el eje horizontal y vertical del plano de origen respectivamente y *x3* es el eje perpendicular a los planos origen e imagen.
- El plano de imagen tiene un tamaño finito conocido y se encuentra a una distancia *f*, llamada longitud focal, de la apertura de cámara en el eje *x3.*
- Cualquier objeto que se quiera proyectar consiste en una serie de puntos donde cada punto *i* se encuentra en la coordenada $(x1, x2, x3)_i$.
- Las proyecciones en el plano de imagen tienen su propio sistema de coordenadas (*y1*, *y2*), donde *y1* representa el eje horizontal e *y2* el eje vertical. El origen de coordenadas se halla en la intersección del plano con el eje *x3*.

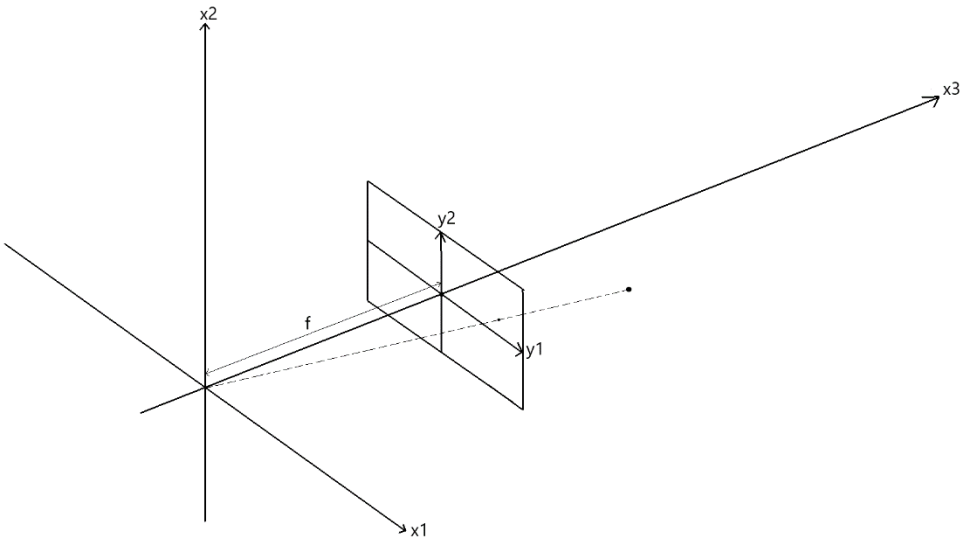

Figura 2.7. Modelo de cámara estenopeica utilizado (Fuente: propia)

El cálculo de las coordenadas de proyección (*y1, y2*) de un punto a partir de las coordenadas (*x1, x2, x3*) de dicho punto es inmediato en este modelo. Utilizando el teorema de Tales se tiene que $\frac{y1}{f} = \frac{x1}{x3}$ y, del mismo modo, $\frac{y2}{f} = \frac{x2}{x3}$, por lo que

$$y1 = f\frac{x1}{x3} \quad (2.1) \quad \text{y} \quad y2 = f\frac{x2}{x3} \quad (2.2.).$$

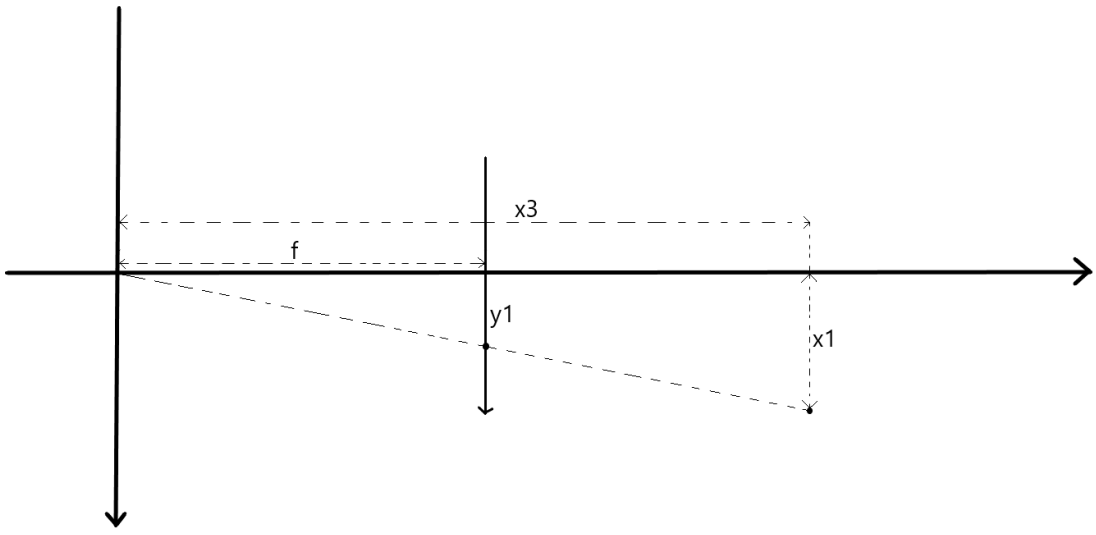

Figura 2.8. Proyección en el plano (x1, x3) (Fuente: propia)

Si se quiere obtener las coordenadas 3D del punto a partir de la proyección 2D, nos queda

$$x1 = x3 * \frac{y1}{f} \quad (2.3) \quad x2 = x3 * \frac{y2}{f} \quad (2.4) \quad \text{y} \quad x3 = f\frac{x1}{y1} = f\frac{x2}{y2} \quad (2.5).$$

Estas ecuaciones, en su forma actual, no se pueden resolver, pues estamos intentando despejar tres incógnitas (*x1*, *x2* y *x3*) con solo dos variables conocidas (*y1* e *y2*), lo que da lugar a una indeterminación. Debido a este problema, muchos sistemas de localización usan cámaras estereoscópicas (con dos o más sensores), sensores de infrarrojos y/o LIDAR (Light Detection and Ranging) para obtener información extra. Nuestro sistema, como se discutirá en el capítulo tres, no necesita de tales añadidos, reduciendo considerablemente su coste.

Capítulo 3. Desarrollo del sistema

En este capítulo se expondrán los pasos tomados para construir nuestro programa. Empezaremos con el entrenamiento de la red neuronal YOLOv4 para detectar nuestro conjunto de conos. Después, integraremos la red para su despliegue dentro de un programa que hace uso de la API openCV. Por último, se discutirá el algoritmo de estimación utilizado y su implementación.

3.1. GPU vs FPGA

Cuando este proyecto fue concebido, la idea inicial era usar un ordenador personal o PC (Personal Computer) solo para desarrollo y depuración, siendo el despliegue final del sistema sobre una FPGA (Field Programmable Gate Array). Las razones exhibidas en aquel entonces eran:

1. Gran potencia de cómputo en paralelo, como las GPU.
2. Orientadas a uso específico, consiguiendo una mayor eficiencia que una plataforma tipo PC.
3. En principio, más baratas que un PC completo.
4. Disponibilidad de modelos orientados a automoción.

Sin embargo, a medida que se ha progresado en el proyecto, nos hemos encontrado repetidas veces con la imposibilidad de llevar a cabo dicha implementación. Se iba a desplegar el sistema en una placa de desarrollo ZedBoard que incorporaba, en un solo chip, una FPGA y un System on Chip (SoC) ARM. Para ello, existían dos métodos para la detección de objetos:

1. Red neuronal YOLO: Xilinx, el fabricante de la FPGA integrada en la placa Zedboard, ofrece software para compilar una red neuronal y desplegarla en lo que ellos llaman una Unidad de Procesamiento de *Deep Learning* o DPU (Deep Learning Processing Unit), una IP (Intelectual Property) sintetizable de Xilinx orientada a la ejecución de redes CNN.

2. Clasificadores en cascada: antes de la popularización de las redes neuronales, se utilizaban clasificadores en cascada para la detección de objetos. La API openCV les daba soporte hasta la versión 3.4. Para su despliegue en una FPGA, la ETSI de Telecomunicación de la Universidad de Málaga había desarrollado un CORE de alto nivel que permite implementar un clasificador sobre una FPGA en tiempo real.

Se ha intentado realizar la implementación de ambas formas, pero ninguna salió adelante por distintos motivos. En el caso de la red neuronal, el compilador exige que la red esté descrita en formato Caffe o Tensorflow, así que no es compatible con YOLO en principio. Sin embargo, Xilinx ofrecía ejemplos de redes basadas en YOLO usando una herramienta de conversión de YOLO a Caffe que, por alguna razón, fue eliminada en algún momento de 2019, así que dejó sin soporte a redes YOLO. Se ha intentado convertir la red a Caffe con otras herramientas disponibles en Internet, pero o no funcionaban o no eran compatibles con el compilador al usar capas personalizadas no presentes en el repositorio Caffe original. En cuanto al clasificador, no se consiguió entrenar un clasificador efectivo usando el mismo *set* de imágenes que la red. Estos intentos han costado varios meses de trabajo.

Cuando finalmente se decidió implementar el sistema en un PC con GPU Nvidia, se descubrió que tenía ciertas ventajas:

1. En la FPGA se habría usado un sensor CMOS, lo que implica diseñar todo el cableado y el *driver*. En un PC solo es necesario enchufar una *webcam* USB.
2. La comunidad de openCV dejó de lado el uso de clasificadores en favor de redes neuronales para la detección de objetos. Debido a ello, hay más ayuda disponible para el uso de estos últimos.
3. Una red neuronal decente tendrá siempre más precisión que un clasificador. El clasificador trabaja con imágenes en blanco y negro, mientras que la red usa imágenes a color. Dado los colores llamativos de los conos, es una ventaja importante. Con esto, también nos ahorramos el escribir un algoritmo de clasificación de conos.
4. El mayor coste del sistema se compensa con mayor flexibilidad y facilidad de implementación, lo que ayuda a reducir los tiempos de desarrollo.

3.2. Entrenamiento de la red neuronal

Lo primero que hay que señalar es que las redes YOLO no usan un *framework* de *machine learning* común como *Caffe* o *Tensorflow*. En su lugar, sus

desarrolladores escribieron un programa para su entrenamiento y evaluación desde cero en C y CUDA llamado *darknet*. Por suerte, como se explicó antes, openCV soporta redes YOLO, aunque de momento no es necesario.

El primer paso del entrenamiento es instalar *darknet*. El código fuente y todas las instrucciones de instalación y entrenamiento se encuentran en un repositorio de Github [5]. La instalación se ha realizado en un PC con las siguientes características:

- Portátil MSI gl62m 7rdx
- Procesador Intel Core i5 7300HQ a 2.5GHz, cuatro núcleos y cuatro hilos
- 8GB de memoria RAM DDR4-2400 monocanal
- GPU Nvidia Geforce GTX 1050 a 1354MHz con 2GB de VRAM GDDR5
- Sistema operativo Ubuntu 18.04
- API CUDA 10.2
- cuDNN 8.0.2
- openCV 4.2.0

El uso de una GPU Nvidia y CUDA es obligatorio. Es posible hacer el entrenamiento sobre una CPU, pero el tiempo requerido no es nada práctico. Al inicio del proyecto, una iteración de entrenamiento de YOLOv3 tardaba unos 30 minutos, mientras que la misma iteración en GPU toma unos 30 segundos, por lo que la GPU es unas 60 veces más rápida que la CPU para este cometido. cuDNN es un conjunto de librerías CUDA orientadas a redes neuronales que mejoran el rendimiento del programa. La API openCV se ha instalado para ofrecer al programa compatibilidad con distintos formatos de imagen, video y entradas de video. Por último, se descarga y se compila *darknet* siguiendo las instrucciones del repositorio, asegurándose de que se compila con CUDA, openCV y, opcionalmente, cuDNN.

Una vez que el programa *darknet* se encuentra instalado y funciona, son necesarios 3 elementos para entrenar una red: un *dataset* de imágenes etiquetadas, un fichero .cfg con la topología de la red neuronal y un fichero .weights con pesos iniciales para el aprendizaje.

El *dataset* consiste en una colección de imágenes “etiquetadas”, donde en cada una se ha señalado manualmente los conos que hay presentes, su tipo y sus bordes. Esto le indica a la red cómo debería funcionar. Las imágenes de nuestro *dataset* se han recogido de fotografías de competiciones MART centradas en la pista de carreras delimitadas por los conos reglamentarios, así como fotos propias con nuestros propios conos, que son copias lo más fielmente posibles a los conos reglamentarios, pues estos eran imposibles de comprar en España. Las etiquetas consisten en ficheros de texto .txt con el mismo nombre que la foto que etiqueta. Cada línea en el fichero representa un objeto deseado a detectar y presenta el siguiente formato:

`<tipo_objeto> <centro_x> <centro_y> <ancho_x> <ancho_y>`

1. `<tipo_objeto>` es un número entero entre 0 y N-1, donde N representa el número de objetos a detectar. Se ha tomado como convención que:
    - Un cono azul es tipo 0
    - Un cono amarillo es tipo 1
    - Un cono naranja chico es tipo 2
    - Un cono naranja grande es tipo 3
2. `<centro_x>` representa la coordenada horizontal del centro del objeto dentro de la imagen.
3. `<centro_y>` representa la coordenada vertical del centro del objeto dentro de la imagen.
4. `<ancho_x>` indica el ancho del rectángulo que envuelve al objeto.
5. `<ancho_y>` indica el alto del rectángulo que envuelve al objeto.

`<centro_x>`, `<centro_y>`, `<ancho_x>` y `<ancho_y>` son números reales entre cero y uno, uno inclusive, que representan el píxel o número de píxeles normalizados respecto al ancho o alto de la imagen en píxeles.

Como etiquetar las imágenes escribiendo texto sería demasiado tedioso, se ha utilizado un programa de etiquetado llamado labelImg. Este programa permite etiquetar una colección de imágenes de forma sencilla y precisa y genera de forma automática los ficheros con el formato de etiquetado de *darknet*.

Figura 3.1. Interfaz de usuario de labelImg

Una vez que se tienen los ficheros de etiquetas, hay que listar de alguna forma las imágenes y etiquetas que forman parte del *dataset* y si se van a usar para entrenamiento o pruebas. En este proyecto no se han hecho pruebas con imágenes etiquetadas, así que todas las imágenes utilizadas se han recogido en un fichero train.txt que incluye la ruta de cada archivo de imagen relativa al ejecutable *darknet*. También se puede añadir una imagen usando la ruta absoluta. También se ha creado un fichero test.txt en blanco, donde deberían estar las imágenes etiquetadas para pruebas.

Conviene crear otro fichero de texto que contenga los nombres de los objetos, ya que hasta ahora el programa solo los identifica con un número entero. Este fichero, que hemos llamado conos.names, incluye en cada línea el nombre del objeto correspondiente al número de línea que ocupa. En nuestro caso, el fichero contiene:

```
cono_azul

cono_amarillo

cono_naranja_chico

cono_naranja_grande
```

Por último, hace falta un fichero de configuración con la información restante y que haga referencia a los demás ficheros creados. Éste es nuestro fichero conos.data, que contiene lo siguiente:

```
classes = 4

train = train.txt

valid = test.txt

names = conos.names

backup = backup
```

La línea *classes* indica el número de objetos a detectar. Como tenemos cuatro conos, son cuatro clases a detectar. La línea *train* señala a nuestro fichero train.txt que incluye el *dataset* de entrenamiento. La línea *valid*, similarmente, apunta a un fichero que contiene el *dataset* para probar la red. En nuestro caso, apunta al fichero vacío test.txt. La línea *names* apunta a nuestro fichero conos.names con los nombres de los objetos. Por último, la línea *backup* apunta a una carpeta donde se guardarán los ficheros .weights con los pesos que se vayan generando a medida que progrese el entrenamiento. En este caso, hay que crear de antemano la carpeta *backup* en el directorio de *darknet.*

El siguiente elemento necesario es el fichero .cfg con la topología de la red neuronal a entrenar. Nosotros hemos seguido la guía publicada en el repositorio de la aplicación: se ha usado una versión modificada del fichero yolov4-tiny-custom.cfg. Siguiendo dicha guía, se ha copiado como yolov4-conos-tiny y se han introducido los siguientes cambios:

1. Se han comentado con una almohadilla las líneas tres y cuatro y se han descomentado las líneas seis y siete. Así, *batch* ahora vale 64 y *subdivisions* se y *subdivisions* 1.
2. Se cambia *subdivisions* a 16. Debido a la memoria limitada de nuestra GPU, para que el entrenamiento pudiera ejecutarse, se elevó a 32.
3. En la línea 20, se ha cambiado *max_batches* a 8000. Esta variable indicará el número máximo de iteraciones a realizar durante el entrenamiento. El valor escogido viene de multiplicar 2000 por el número de objetos, cuatro en nuestro caso.

4. En la línea 22 se ha cambiado *steps* a 6400, 7200. Estos son el 80% y el 90% de *max_batches* respectivamente.
5. En las líneas ocho y nueve, se ha configurado *width* y *height* a 416. Estas dos variables indican el tamaño de la imagen de entrada a la red. Un valor más alto suele ofrecer mejor resultado a costa de un peor rendimiento. Es importante que el valor sea múltiplo de 32.
6. En las líneas 220 y 269, se ha de cambiar el valor *classes* a 4.
7. En las líneas 212 y 263, se ha cambiado el valor de *filter* a 27. Éste es el resultado de $(N + 5) \cdot 3$, siendo N el número de clases.

Por último, necesitamos un fichero .weights con unos pesos iniciales. Por suerte, solo hay que descargar el fichero yolov4-tiny.conv.29 del repositorio. Este fichero, aun sin compartir la extensión, es un fichero de pesos iniciales.

```
Carpeta de trabajo
│   darknet
│   yolov4-conos-tiny.cfg
│   yolov4-tiny.conv.29
│   conos.data
│   conos.names
│   test.txt
│   train.txt
│
└───Muestras
    │   Foto1.jpg
    │   Foto1.txt
    │   ...
    │   Foton.jpg
    │   Foton.txt
```

Figura 3.2. Estructura de directorio para entrenamiento con n imágenes

Una vez que tenemos todos preparado, podemos lanzar el proceso de entrenamiento con el comando:

```
./darknet detector train conos.data yolov4-conos-tiny.cfg yolov4-tiny.conv.29
```

Si todo va bien, aparecerá una ventana con una gráfica. Dicha gráfica mostrará las pérdidas de la red neuronal con cada iteración. Una red funcional puede presentar un valor de pérdidas de entre 0.05 en casos simples y 3.0 en casos

muy complejos. También sirve para saber cuándo dejar de entrenar, ya que llegado a cierto punto, las pérdidas dejan de bajar y comienzan a oscilar. En la figura 3.2 se muestra la gráfica de pérdidas resultante de nuestra red:

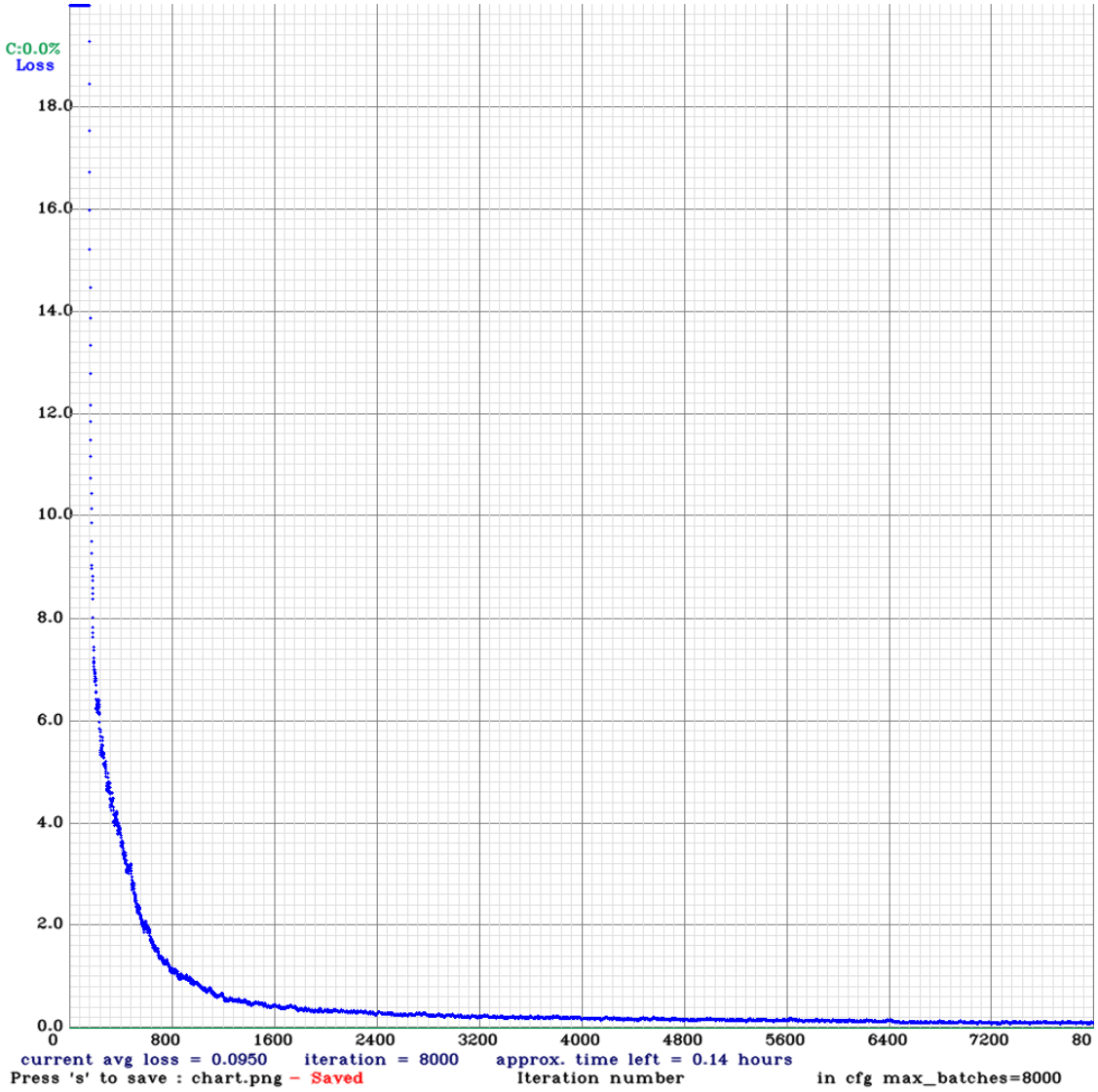

Figura 3.3. Gráfica de pérdidas en el entrenamiento de yolov4-conos-tiny

Se puede ver que las pérdidas de nuestra red son muy bajas. Sin embargo, hay que tener cuidado, pues un resultado muy bajo puede ser producto de un sobreajuste, esto es, la red funciona muy bien con el *set* de entrenamiento, pero no sirve en otros casos. Esto se tendrá que comprobar con imágenes fuera del *dataset*.

El proceso de entrenamiento puede ser bastante largo, de varias horas, dependiendo del tamaño del *dataset* y la capacidad de la GPU utilizada. En

nuestro caso, con un *dataset* de 29 imágenes y una GTX 1050 se han tardado unas 12 horas en completar las 8000 iteraciones calculadas antes.

En las figuras 3.3, 3.4 y 3.5 se muestran algunos resultados de la red entrenada. Para ello, se ejecuta el comando

```
./darknet detector test conos.data yolov4-conos-tiny.cfg
yolov4-conos-tiny.weights
```

Una vez dentro del programa, se pedirá la ruta a un fichero de imagen o video. Si le proporcionamos uno, ejecutará la red sobre el fichero y, una vez que se tienen los resultados, se mostrarán por pantalla tanto en la consola como sobre la imagen y se guardará en un fichero *predictions*. También se muestra por consola el tiempo de ejecución. Para que funcione en tiempo real, la red ha de tardar como máximo 30 milisegundos.

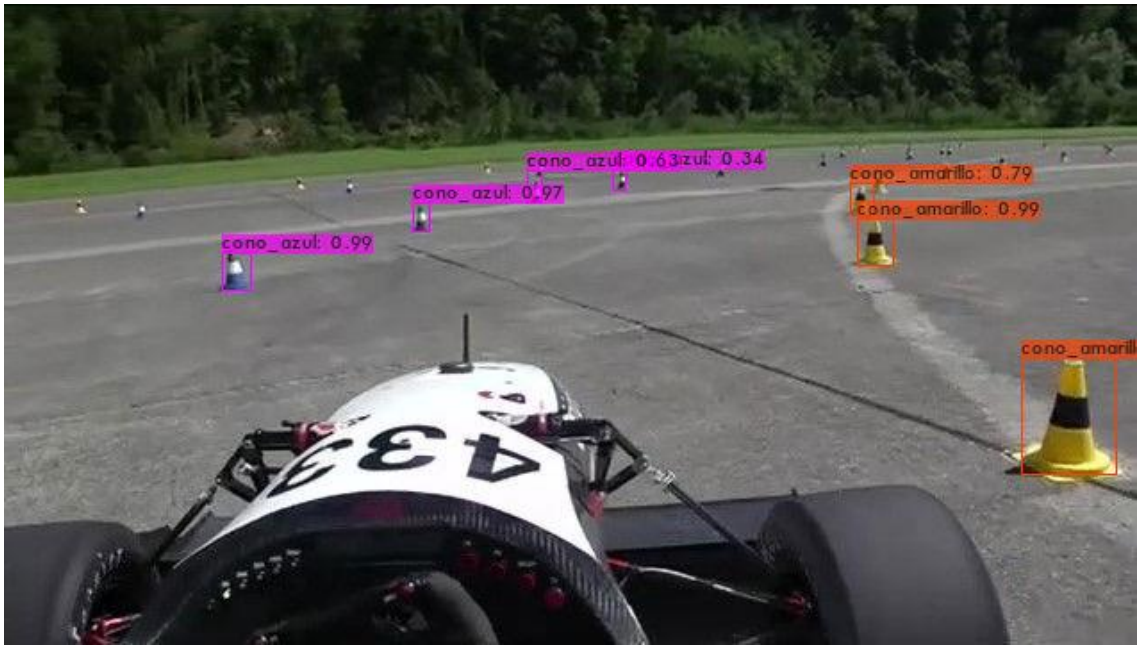

Figura 3.4. Primera prueba de la red neuronal en *darknet*

Figura 3.5. Segunda prueba de la red neuronal en *darknet*

Figura 3.6. Tercera prueba de la red neuronal en *darknet*

En la figura 3.3, vemos que no se detectan los conos que están más lejos. Esto es de esperar por dos razones: la limitada resolución de la imagen y el *blob* (matriz de cuatro dimensiones que contiene una o varias imágenes y sirve de entrada a la red, se explicará con más detalle en el siguiente apartado) y el hecho de que, a partir de cierta distancia, no se etiquetaron los conos del *dataset* para evitar posibles confusiones durante el entrenamiento. Respecto al tiempo de ejecución, en los tres se ha tardado unos 25 milisegundos, así que cumple la condición de tiempo real.

3.3. Detección de conos en una imagen

Una vez que tenemos una red neuronal funcional, el siguiente paso es integrarla en un programa que pueda extraer la información proporcionada por ella para otros usos. Para ello, usaremos openCV, pues permite implementar la red y procesar la información de salida de forma sencilla. Además, openCV ya incluye un programa de ejemplo de detección de objetos usando YOLO en el fichero object_detection.cpp, sobre el cual nos basaremos.

Un paso importante, y que a menudo se olvida, es modificar el fichero .cfg para inferencia en lugar de entrenamiento. Para ello, se tiene que deshacer paso 1 de las modificaciones hechas a dicho fichero, esto es, las variables *batch* y *subdivisions* tienen que configurarse a valor 1.

Como la idea es incluir la red dentro de un programa mayor, todo el código relacionado con la red y la información de salida se ha escrito dentro de una función de C++ llamada *detectar_conos*. Dicha función requiere como entradas un elemento tipo *Mat* con la imagen a procesar y otro objeto tipo *Net* que contenga nuestra red neuronal. Como resultado, devuelve un contenedor tipo vector de tamaño variable donde cada elemento es una estructura creada por nosotros llamada *Conos_detectados*, que incluye:

1. Un entero llamado *tipo* que identifica el tipo de cono siguiendo la convención de etiquetado explicada en el apartado 3.1.
2. Un flotante llamado *confianza* que indica la probabilidad de que la predicción de este cono sea real. Sirve para descartar falsos positivos.
3. Un objeto tipo *Rect* llamado posición. Este tipo de objeto apunta a un área rectangular dentro de la imagen y marca los bordes del cono.

La cabecera de la función queda así:

```
vector<Conos_detectados> detectar_conos(Mat &imagen, Net &red, float umbral_confianza, float umbral_NMS)
```

Antes de entregar la imagen a la red para que la procese, es preciso convertirla en un “*blob*”, que es lo que las redes neuronales piden como entrada. Un *blob* consiste en una matriz de cuatro dimensiones, donde las dimensiones son:

1. El lote: denomina a una imagen dentro de un conjunto que están contenidas en el mismo *blob*. Si solo es una imagen, la dimensión lote tiene tamaño unidad.
2. Los canales: puede tener tamaño uno, tres o cuatro dependiendo de si la imagen es en escala de grises, a color o a color con canal alfa respectivamente.
3. La altura de la imagen.
4. La anchura de la imagen.

Además, la conversión de imagen permite realizar ciertos cambios, como cambio de resolución, sustracción de una media, escalado e intercambio de los canales R (rojo) y B (azul). En nuestro caso, se ha redimensionado a la resolución requerida por la red de 416×416 y se han intercambiado los canales R y B debido a que las redes YOLO usan el esquema BGR. Cada elemento del *blob* consiste en un entero sin signo de ocho bits.

Una vez que tenemos el *blob* preparado, se lo pasamos a la red. Cuando introducimos el *blob*, también escalamos el valor de cada elemento multiplicándolo por 0.00392, que es 1/255. De este modo, el entero de ocho bits pasa a ser un flotante con un valor entre 0 y 1, ambos inclusive. Finalmente, se puede ejecutar la red hasta la última capa.

El programa de ejemplo *object_detection.cpp* acepta dos tipos de salida de red neuronal, una de tipo detección y otra de tipo región. Las redes YOLO devuelven esta última. Una salida tipo región consiste en una matriz donde cada fila corresponde a un posible objeto detectado. Las columnas están compuestas por cuatro flotantes indicando la posición con el mismo esquema que las etiquetas del apartado 3.1 más un flotante por cada clase de objeto a detectar. Dichos flotantes indican la probabilidad de que el objeto detectado corresponda a cada una de las posibles clases, por lo que se buscará aquella clase con mayor probabilidad.

Tras la ejecución de la red y la obtención de los datos, se procesa cada posible objeto detectado realizando las siguientes operaciones:

1. Se busca la clase de objeto con mayor puntuación para obtener el tipo de objeto y la confianza.

2. Si la confianza es menor al umbral establecido, se descarta el objeto.
3. Si, por el contrario, es mayor, se convierten los datos de posición para crear el objeto *Rect* que señala la ubicación del cono en la imagen.

Por último, antes de devolver los conos detectados al programa llamante, es necesario eliminar posibles duplicados. Esto se hace con el algoritmo de Supresión de No Máximos o NMS (Non Maximum Suppression), el cual es incluido por openCV. Para ejecutar el algoritmo, es necesario establecer un umbral NMS que indique cuánto solapamiento han de tener dos objetos antes de que se consideren duplicados. Por defecto, se usa un umbral NMS de 0.4.

Una vez descartados los duplicados, se añade la lista de conos al vector de *Conos_detectados* y se sale de la función.

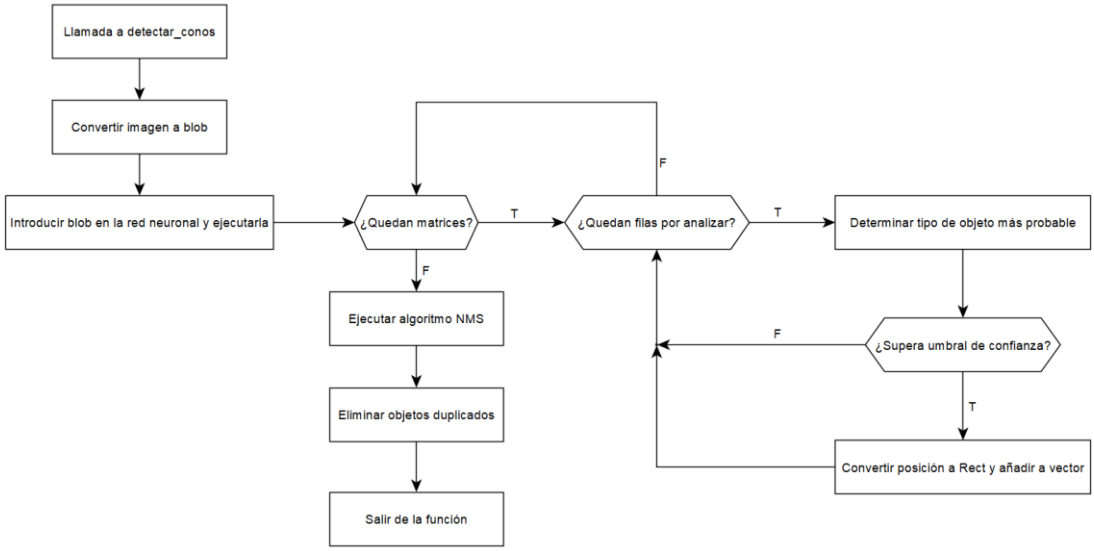

Figura 3.7. Flujograma de la función detectar_conos

3.4. Posicionamiento

En este apartado, se discutirá en primer lugar el algoritmo usado y, posteriormente, su implementación software.

Como se vio en el apartado 2.5, la relación entre un punto $(x1, x2, x3)$ y su representación en una imagen $(y1, y2)$ venía dada por las ecuaciones 2.1 y 2.2. Si se dividen entre sí, nos queda $\frac{y1}{y2} = \frac{x1}{x2}$ (3.1). De esta ecuación se puede deducir

que las proporciones del objeto real se mantienen en la imagen, sin importar su distancia a la cámara ni la longitud focal de ésta.

Conocemos las dimensiones de los conos y, suponiendo una detección perfecta por parte de la red neuronal, sus dimensiones en la imagen son proporcionales. Sea W el ancho real del cono, w el ancho que presenta en la imagen e *y1* y *x1* la coordenada horizontal del borde izquierdo en la imagen y en el mundo real respectivamente, se puede expresar el borde derecho como la sustitución de la ecuación (2.1) $y1 + w = f\frac{x1+W}{x3}$ (3.2). Si se divide por la ecuación (2.1), nos da

$$\frac{y1+w}{y1} = \frac{x1+W}{x1} \rightarrow \cancel{x1 \cdot y1} + x1 \cdot w = \cancel{y1 \cdot x1} + y1 \cdot W \rightarrow x1 = y1\frac{W}{w} \text{ (3.3).}$$

La ecuación 3.3. nos resuelve la coordenada horizontal del cono. Si se inspecciona un poco, vemos que tiene sentido ya que, a medida que se aleja el cono, se ve de menor tamaño en la imagen. Además, para mantener una posición horizontal concreta en la imagen al alejarse, también se ha de alejar del centro de la imagen en la misma medida.

La posición vertical *x2* es irrelevante para nuestro propósito, ya que el vehículo busca un mapeo 2D de la pista y, además, sabemos que todos los conos están en el suelo a una altura concreta dependiendo de la elevación de la cámara. De todos modos, su ecuación se obtendría simplemente sustituyendo 3.3 con las componentes verticales. Siendo H la altura real del cono y h su altura en la imagen, la ecuación de *x2* nos queda como

$$x2 = y2\frac{H}{h} \quad \text{(3.4).}$$

Nos queda la coordenada de profundidad *x3*. Para ello, se puede sustituir la ecuación 2.5 con los resultados de las ecuaciones 3.3 y 3.4, dando como resultado

$$x3 = f\frac{\cancel{y1}\frac{W}{w}}{\cancel{y1}} = f\frac{W}{w} \quad \text{(3.5)} \quad \text{y} \quad x3 = f\frac{\cancel{y2}\frac{H}{h}}{\cancel{y2}} = f\frac{H}{h} \quad \text{(3.6).}$$

Para compensar de forma parcial posibles errores en una de las dos coordenadas, se ha decidido utilizar la media de ambas medidas:

$$x3 = f\frac{\left(\frac{W}{w}+\frac{H}{h}\right)}{2} \text{ (3.7).}$$

Una vez que se ha resuelto el problema de la posición, hay que escribirlo en C++. Para ello, se ha escrito otra función que acepta como entrada:

1. El vector de *Conos_detectados* obtenido anteriormente.
2. La imagen de donde se han obtenido.
3. Una estructura *Param_calibracion* con varios parámetros en coma flotante para calcular y corregir la posición. Se discutirá luego.

Como resultado, devuelve un vector de una estructura tipo *Conos_localizados*, que es similar a *Conos_detectados* pero en vez del objeto *Rect* tiene dos variables tipo *float*, *x1* y *x3*, con la posición estimada. La cabecera de la función queda como:

```
vector<Conos_localizados>     estimar_posicion(const     vector<Conos_detectados>
conos_detectados, const  Mat imagen, const Param_calibracion p)
```

La función consiste en un bucle que realiza el cálculo para cada cono del vector. En primer lugar, se copia la información común a los dos tipos de estructuras, es decir, el tipo de cono y la confianza. Como el objeto *Rect* devuelve las coordenadas en forma de píxel, hay que normalizarlo. Aquí se ha decidido que la imagen estará normalizada a una altura de un metro con el origen en su centro. Esto significa que, si la imagen tiene un formato panorámico 16/9, la imagen normalizada tendrá unas dimensiones de 1,77 metros. Con esto aclarado, ahora se convierte el ancho, el alto y las coordenadas del centro del cono a este sistema de referencia normalizado.

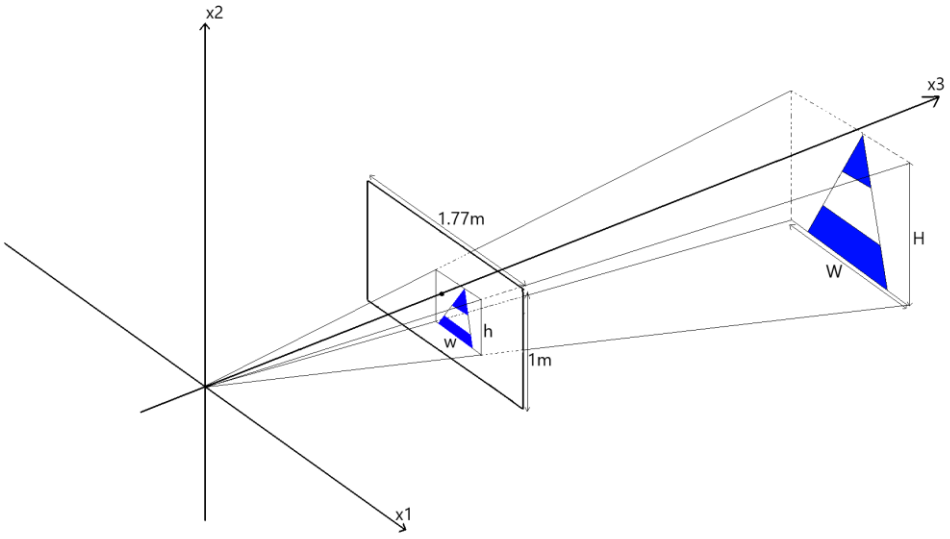

Figura 3.8. Modelo de cámara estenopeica para posicionamiento de conos

Como el cono naranja tiene un ancho y alto diferentes a los demás, se aplican las ecuaciones (3.3) y (3.7) con distintos valores de W y H dependiendo de si el cono es de tipo tres o no. La longitud focal es uno de los valores de calibración pasados en la estructura *Param_calibracion*. El resto de parámetros (*a1, b1, c1, a3, b3* y *c3*) son usados para definir dos polinomios de grado dos con los que corregir posibles errores constantes en las medidas. Las operaciones realizadas son

$$cono.x3 = c3 \cdot x3^2 + b3 \cdot x3 + a3 \quad (3.8)$$

$$cono.x1 = x1 + c1 \cdot x3^2 + b1 \cdot x3 + a1 \quad (3.9)$$

Esta corrección se realiza porque se observó empíricamente que, a mayor distancia del cono respecto a la cámara, mayor error se comete en la posición. Hemos considerado que un ajuste cuadrático es suficiente. Por último, se añade la estructura ya rellena al vector. Al terminar con todos los conos, termina la función.

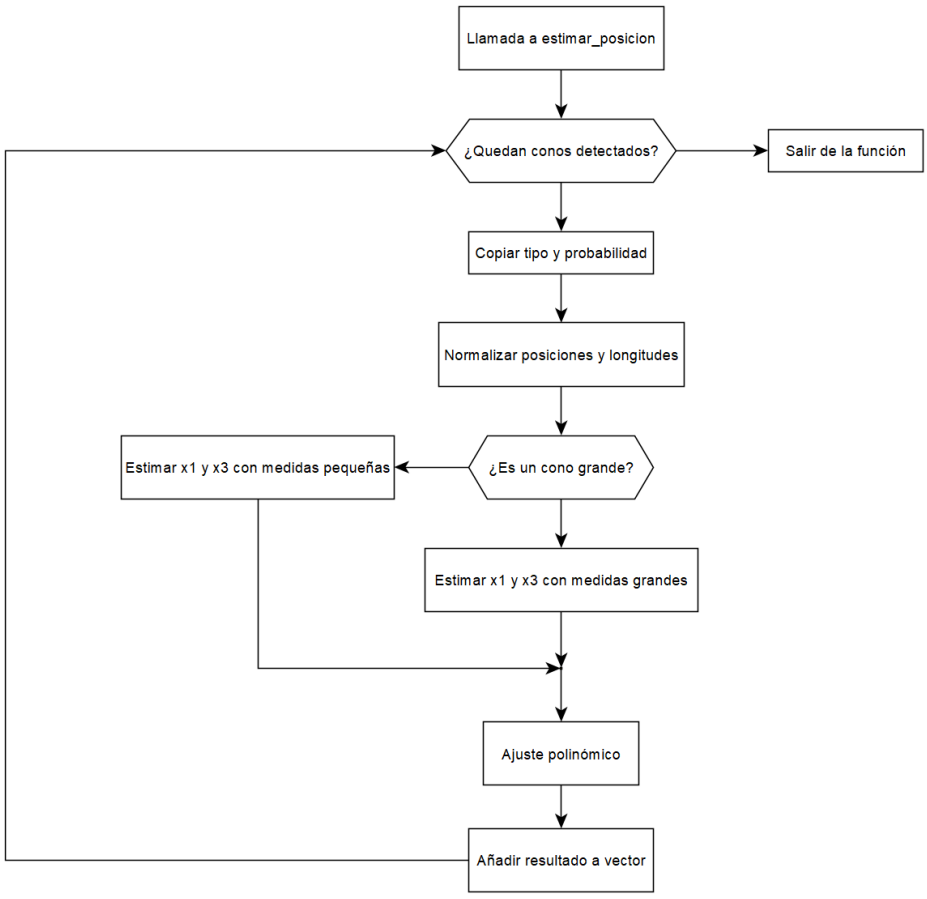

Figura 3.9. Flujograma de la función estimar_posicion

Capítulo 4. Demo, pruebas y resultados

En este apartado se expondrá, en primer lugar, el programa *main.cpp* escrito para evaluar nuestras funciones de detección y posicionamiento de conos y que, además, incluye otras funciones necesarias, como la calibración de cámara. Después se mostrarán las pruebas hechas junto con sus resultados.

4.1. main.cpp

Todas las estructuras y funciones descritas en el capítulo 3 se han escrito en dos ficheros conos.hpp y conos.cpp para facilitar su integración en otros programas. El fichero conos.hpp incluye, además, las cabeceras *include* necesarias, las dimensiones de los conos para la estimación y la declaración de dos nuevas funciones implementadas en conos.cpp: leer_parametros y calibrar_f. La primera función lee de un fichero de texto cuyo nombre se le pasa como argumento y rellena la estructura Param_calibracion. Devuelve 0 si ha tenido éxito y -1 si ha fallado. Su declaración es:

```
int leer_parametros(string nombre_fichero, Param_calibracion &parametros)
```

La segunda función toma como parámetros la entrada de imágenes, la red YOLO, los umbrales de confianza y NMS y Param_calibracion y devuelve la longitud focal estimada a partir de la imagen, suponiendo que en ella hay un cono exactamente a un metro de la cámara. En primer lugar, extrae imágenes de la entrada y las procesa por la red hasta encontrar al menos un cono. Después, usando el primer cono encontrado, aplica una variante invertida de la ecuación 3.7 para obtener f. Dicha variante de la ecuación se desarrolla como sigue:

$$x3 = f\frac{\left(\frac{W}{w}+\frac{H}{h}\right)}{2} = 1 \rightarrow f = \frac{2}{\left(\frac{W}{w}+\frac{H}{h}\right)} \quad (4.1)$$

En su implementación, al igual que en la función estimar_posición, se normalizan las medidas del cono y se estima f según si el cono es grande o chico. Por último, devuelve f y sale de la función. Su declaración queda como sigue:

```
double calibrar_f(VideoCapture cap, Net red, Param_calibracion parametros, float umbral_confianza, float umbral_NMS)
```

Estos dos ficheros forman parte de un programa demo main.cpp que incluye las siguientes funcionalidades:

- Lee de un fichero de imagen o video o cámara a elección del usuario.
- Permite cambiar los ficheros .cfg, .weights y .txt que se leen.
- Se pueden cambiar los umbrales de confianza y NMS.
- Muestra las imágenes por pantalla en tiempo real.
- Resalta los conos en la imagen con un color diferente dependiendo de su tipo:
    - Azul: azul
    - Amarillo: amarillo
    - Naranja chico: naranja
    - Naranja grande: rojo
- Sobre el borde del cono escribe las distancias x1 y x3
- En la esquina inferior izquierda muestra los FPS actual y mínimo.
- Permite hacer capturas de la imagen mostrada.
- Tiene modo de calibración, que calcula la longitud focal y la escribe al fichero de parámetros de calibración.

El programa se ha compilado usando CMake, siguiendo la guía de openCV. Sin embargo, en la línea add_executable del fichero CMakeLists.txt hay que añadir conos.cpp aparte de main.cpp. Suponiendo que los ficheros yolov4-conos-tiny.cfg, yolov4-conos-tiny.weights y calibración.txt (este incluye los parámetros de calibración, un valor por línea) se encuentran en el mismo directorio que el programa y éste se llama det_conos, se puede escribir en la línea de comandos

```
./det_conos -f ruta_fichero
```

Para abrir un fichero de imagen o video, o simplemente

```
./det_conos
```

Para que abra la primera cámara que encuentre en el sistema. Si se llama

```
./det_conos --focal
```

Entrará en el modo de calibración anteriormente explicado. Se recomienda ejecutar el programa antes en modo normal para asegurarse de que se detecta correctamente el cono y no hay otros conos detectados (reales o falsos) que puedan interferir. En este modo, una vez calculado f mostrará por consola el nuevo valor y se cerrará el programa sin mostrar imagen. Si se escribe

```
./det_conos -h
```

Se mostrará en la consola una pequeña ayuda del programa, explicando todos los parámetros que se pueden añadir. Durante el funcionamiento normal, se puede hacer capturas de la imagen pulsando la barra espaciadora. La imagen se guardará en un fichero con nombre *Resultado_X.png*, donde X es un número que empieza en uno y se va incrementando con cada captura. De este modo, se pueden hacer tantas capturas como se consideren necesarias, aunque hay que mover o renombrar las imágenes entre sesiones para evitar sobreescrituras. Por último, si se pulsa q, se cierra el programa.

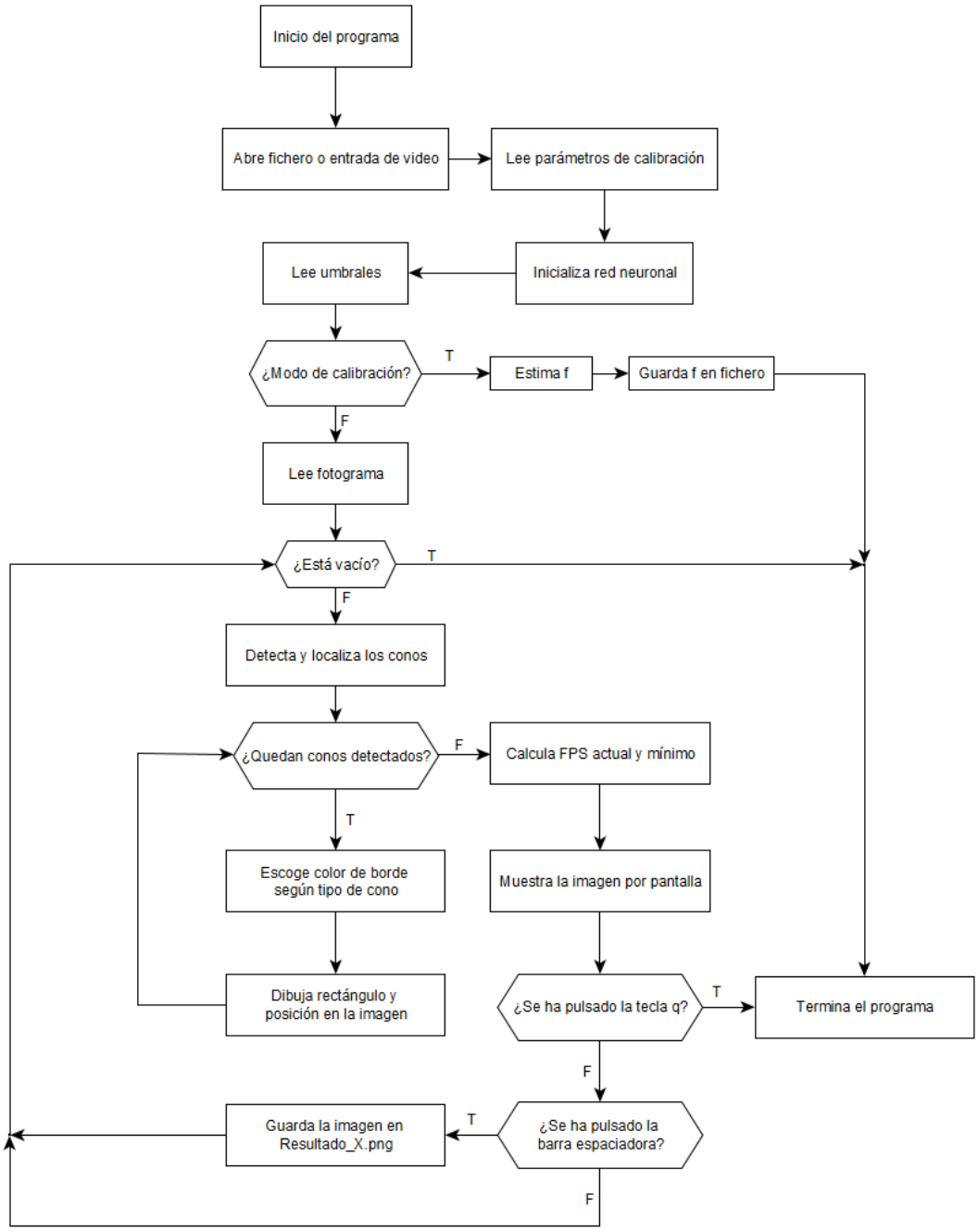

Figura 4.2. Flujograma de main.cpp

4.2. Pruebas

En esta fase del proyecto, se van a hacer dos baterías de pruebas. La primera va a servir para establecer, mediante un ajuste polinómico, los parámetros de calibración que reduzcan el error de medida lo máximo posible. En la segunda, una vez se ha realizado la calibración polinómica, se caracterizará el sistema en los términos de precisión y rendimiento mencionados en la introducción.

Las pruebas se han hecho en el mismo equipo que en el entrenamiento, salvo por una webcam distinta a la incorporada:

- Portátil MSI gl62m 7rdx
- Procesador Intel Core i5 7300HQ a 2.5GHz, cuatro núcleos y cuatro hilos
- 8GB de memoria RAM DDR4-2400 monocanal
- GPU Nvidia Geforce GTX 1050 a 1354MHz con 2GB de VRAM GDDR5
- Webcam Owlotech Start, resolución 1280×720 a 30 FPS
- Sistema operativo Ubuntu 18.04
- API CUDA 10.2
- cuDNN 8.0.2
- openCV 4.2.0

No fue posible obtener ejemplares de los modelos de cono oficiales de la competición, debido a que se venden solo en Alemania y no tienen opción de envío a España. En su lugar, se han comprado conos con un tamaño y pintura lo más similar a los oficiales. Como no se encontraron conos azules ni amarillos, se han comprado tres conos naranjas pequeños y uno grande y se ha pintado dos de los tres conos pequeños. Los conos de prueba resultantes aparecen en las figuras 4.1 y 4.2. Para evitar problemas, se les han hecho diversas fotos y se han incluido en el *set* de entrenamiento.

El primer paso es calibrar la cámara, esto es, estimar su longitud focal f. Para ello, hemos colocado el cono de prueba azul a un metro de la cámara, la cual está colocada a unos 20cm del suelo, y se ha llamado al programa con el modo de calibración. Nos ha dado como resultado una longitud focal de 2.304454 metros. Como curiosidad, durante las primeras pruebas se usó la *webcam* integrada del portátil y ésta tanto como la cámara usada en las figuras 3.3, 3.4 y 3.5 tienen una longitud focal del orden de 80cm. Esto significa que la *webcam* que vamos a usar aquí tiene un campo de visión menor que aquellas dos. Además, antes de calibrar la cámara, el programa presentaba unas distancias demasiado pequeñas, del orden de los milímetros estando el cono a

aproximadamente un metro de la cámara. Esto pone de manifiesto la vital importancia de la calibración.

Las pruebas hechas consisten en una serie de medidas de las posiciones de conos colocados en una variedad de posiciones, tanto en el eje *x1* como en *x3.* Las pruebas se han hecho en el aparcamiento de la ETSI de Telecomunicación de la Universidad de Málaga a plena luz del día para replicar lo más fielmente posible el entorno en el que va a trabajar el sistema, esto es, una pista de carreras. La cámara se ha colocado a 19cm del suelo. En la figura 4.1 se muestra el entorno de pruebas. Como se observa en ella, se han aprovechado las marcas viales para evitar desviaciones.

Figura 4.2. Entorno de pruebas

En la primera ronda de pruebas, se han tomado 16 muestras de conos. Se ha empezado por un cono colocado a un metro enfrente de la cámara y, en cada muestra sucesiva, se ha alejado el cono un metro hasta los 16 metros. La posición *x1* se ha elegido de forma pseudoaleatoria, aunque dentro del campo de visión de la cámara, con un intervalo entre -2 y 1.57 metros a intervalos de 25 cm. El tipo de cono también se ha elegido de manera pseudoaleatoria. En este punto hay que aclarar que la imagen y la detección presentan ruido. Esto significa que, aun manteniendo la cámara y el cono en la misma posición, los bordes detectados y, en consecuencia, la distancia puede variar de un fotograma al siguiente. En las mediciones registradas se ha estimado una media de dichas

distancias. Los resultados de la primera serie de pruebas se muestran en la tabla 4.1. Se considera que la posición de los conos tiene un error de ± 10cm.

Tipo	x1 real (m)	x1 medida (m)	x3 real (m)	x3 medida (m)	Comentarios
Azul	0	0	1	1.15	
Amarillo	0.25	0.2	2	2.1	
Naranja chico	-0.5	-0.47	3	3.5	
Naranja grande	-1	-0.85	4	4.1	
Naranja chico	0.75	0.75	5	5.7	
Azul	1.25	1.24	6	7.5	
Naranja grande	-0.25	-0.27	7	7.5	
Amarillo	-1.5	-1.53	8	9.5	
Amarillo	1	0.9	9	10.2	
Naranja Grande	-2	-1.6	10	8.3	
Azul	-0.75	0.92	11	15.3	
Naranja chico	1.5	…	12	…	(detección inestable)
Naranja grande	-1.75	-1.76	13	14.5	
Naranja chico	-1.25	-1.72	14	25	(detectado como naranja grande)
Amarillo	0.5	0.4	15	20	
Azul	1.75	1.7	16	19	

Tabla 4.1. Resultados de la primera prueba

Para entender mejor los resultados obtenidos, se ha representado el error cometido en ambas medidas en las figuras 4.2 y 4.3.

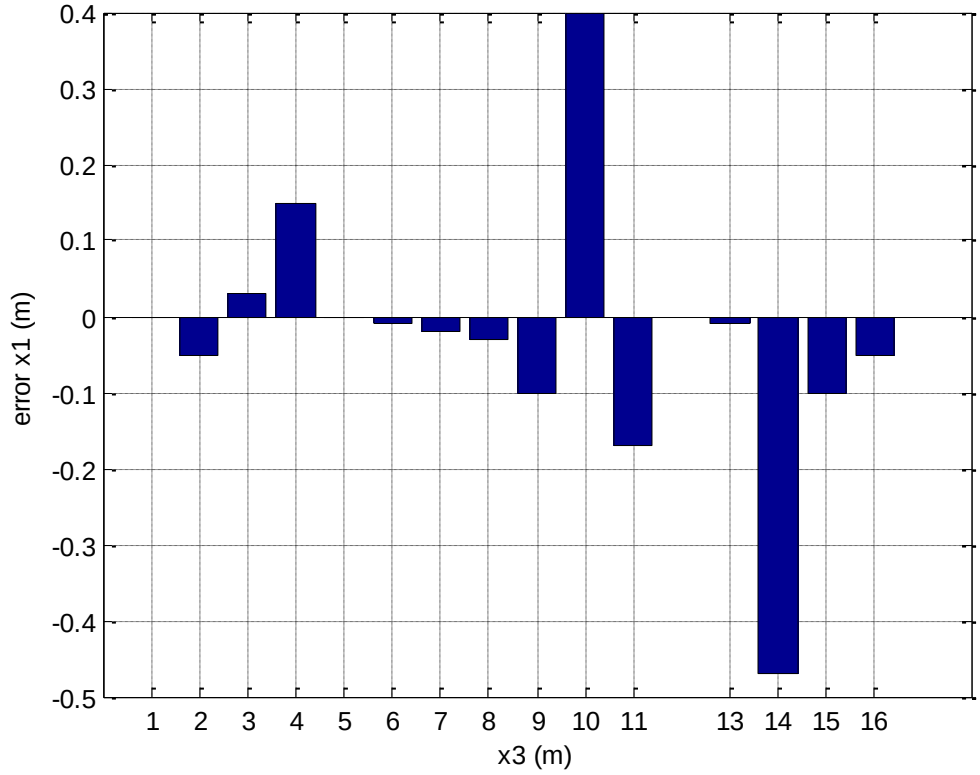

Figura 4.3. Error de medida de x1

En la figura 4.2 observamos que el error en la coordenada *x1* es relativamente pequeño, con más de la mitad de los casos por debajo de los 10cm de error de colocación de los conos. Además, el alto error en la medida 14 es debido principalmente ha que se ha detectado un cono grande en vez de uno chico, alterando las medidas. La medida hecha a diez metros también tiene un error relativamente alto, pero puede deberse a que tanto la distancia *x1* como la *x3* son más o menos grandes, por lo que el error se ha acumulado. Viendo estas medidas, se ha decidido que no es necesario ningún ajuste a *x1*.

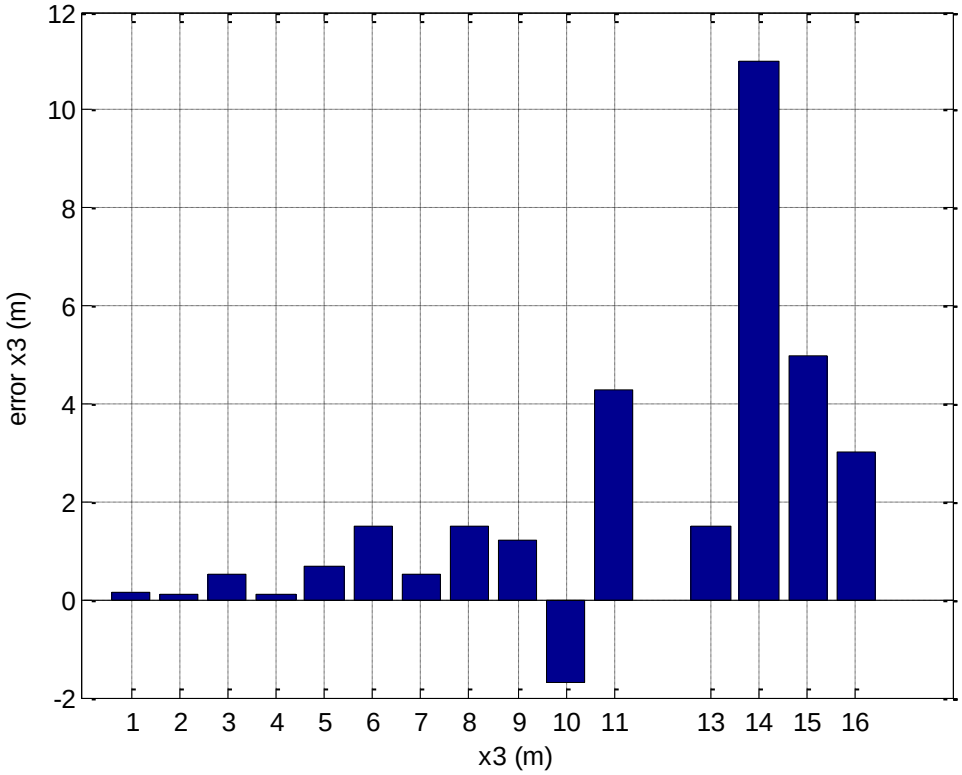

Figura 4.4. Error de medida de x3

En el caso de la medida de *x3*, es prácticamente el caso contrario: el error es relativamente grande. Además, vemos una posible tendencia al alza con la distancia. Siendo éste el caso, quizá se pueda extraer más información en términos de error relativo, el cual es mostrado en la figura 4.4.

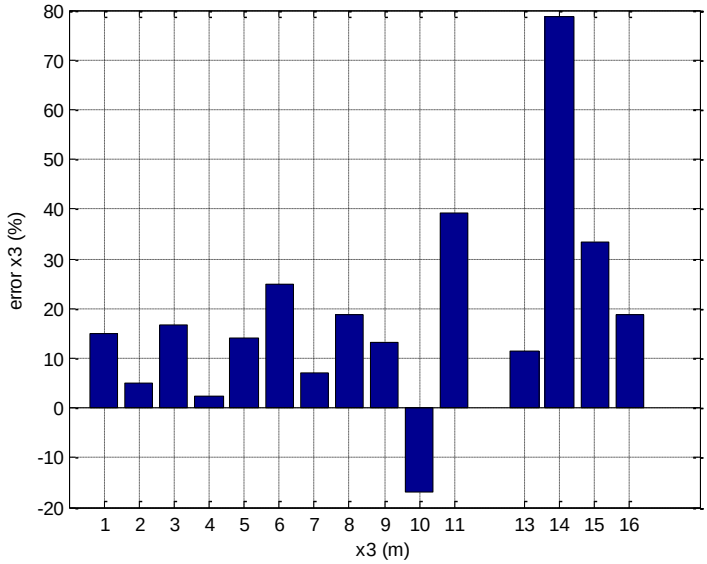

Figura 4.5. Error relativo de medida de x3

Aquí se puede observar que, por regla general, el error está contenido dentro del 20% hasta los diez metros. A partir de los 10 metros, la medida prácticamente carece de fiabilidad.

Por esta razón, se han usados las medidas de uno a diez metros para realizar un ajuste polinómico de orden 2, con los siguientes coeficientes como resultado $a3 = -0.3450, b3 = 1.0927,\ c3 = -0.0149$. Si se aplica el ajuste a las medidas ya hechas y se calcula el error igual que antes, nos quedan las siguientes figuras:

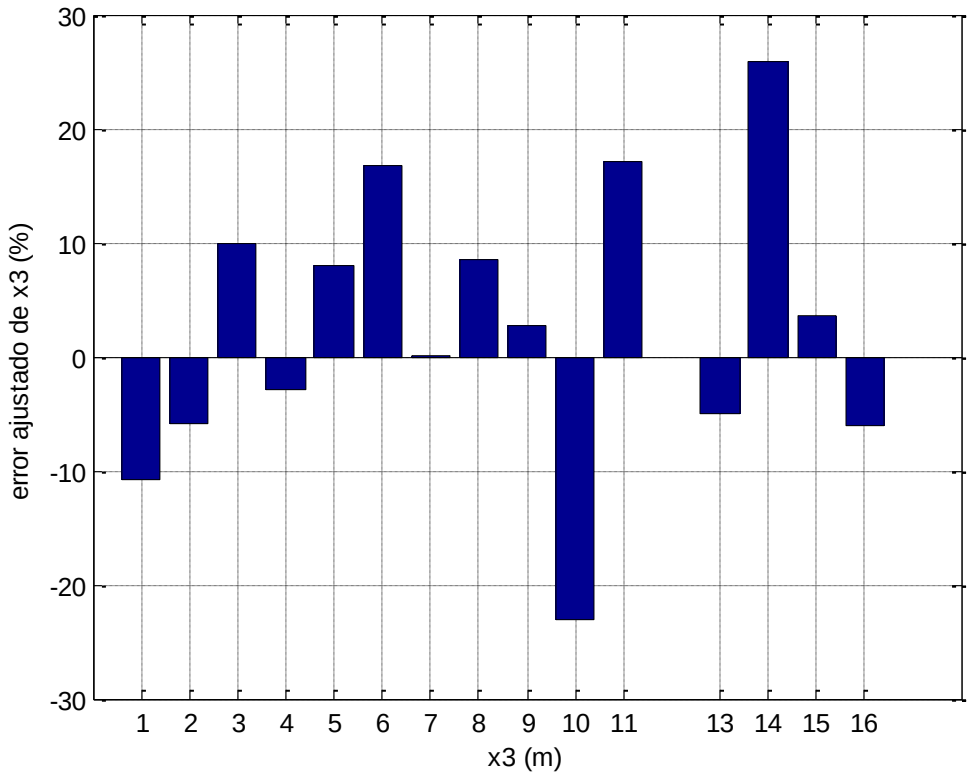

Figura 4.6. Error relativo ajustado de medida de x3

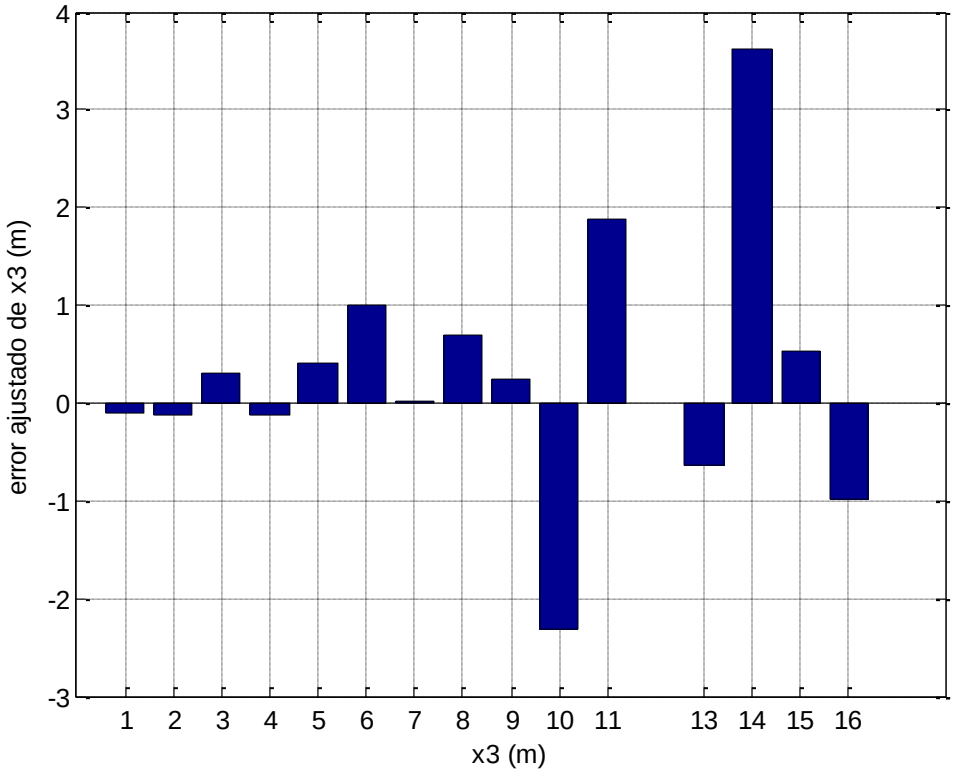

Figura 4.7. Error ajustado de medida de x3

Se puede apreciar que, hasta los nueve metros, el error está acotado por debajo del metro. Para validar este ajuste, es necesario una segunda prueba.

En esta primera prueba, también se ha tomado otras medidas no incluidas en las gráficas. Se trata de una pequeña prueba de alcance, con cuatro muestras desde los 17 hasta los 20 metros. Los resultados se muestran en la tabla 4.2.

Tipo	**x1 real (m)**	**x1 medida (m)**	**x3 real (m)**	**x3 medida (m)**	**Comentarios**
Amarillo	0	-0.13	17	25	Detección inestable. Confundido con naranja chico
Azul	0	-0.14	18	23	
Amarillo	0	…	19	…	No detectado
Azul	0	-0.15	20	24	

Tabla 4.2. Resultados de la prueba de alcance

Ignorando las distancias estimadas, se pueden sacar dos conclusiones: los conos amarillos dejan de detectarse correctamente a esas distancias y los conos azules podrían ser reconocidos incluso más lejos. La posible razón del problema con los conos amarillos es que, a esa distancia y con la calidad de imagen disponible (colores muy claros), empieza a confundirse con el asfalto. Los conos naranjas no se han incluido porque ya empezaban a dar problemas en el entorno de los 14-15 metros.

La segunda prueba se ha hecho en condiciones idénticas a la primera, salvo por el ajuste polinómico de *x3*. Los resultados están reflejados en la tabla 4.3.

Tipo	x1 real (m)	x1 medida (m)	x3 real (m)	x3 medida (m)	Comentarios
Naranja chico	0	0	1	0.96	
Amarillo	-0.25	-0.19	2	2	
Naranja grande	0.5	0.44	3	3.02	
Azul	0.75	0.76	4	4.35	
Amarillo	-0.75	-0.7	5	5.43	
Naranja chico	-1.5	-1.52	6	6.78	
Azul	1.25	1.25	7	7.62	
Naranja grande	-2	-1.94	8	8.17	
Naranja grande	-0.5	-0.5	9	10.2	
Azul	-1.25	-1.47	10	12.1	
Amarillo	1	1.17	11	12.5	
Naranja chico	1.75	1.96	12	13.2	
Azul	-1	-1.06	13	14.2	
Naranja grande	0.25	0.3	14	14.4	
Naranja chico	-1.75	-1.57	15	14.1	
Amarillo	1.5	1.53	16	14.8	

Tabla 4.3. Resultados de la segunda prueba

En la figura 4.7 se muestra el error cometido al medir *x1* en la segunda prueba. Como no se ha hecho ningún ajuste a esta medida, es de esperar que el error esté distribuido de forma similar.

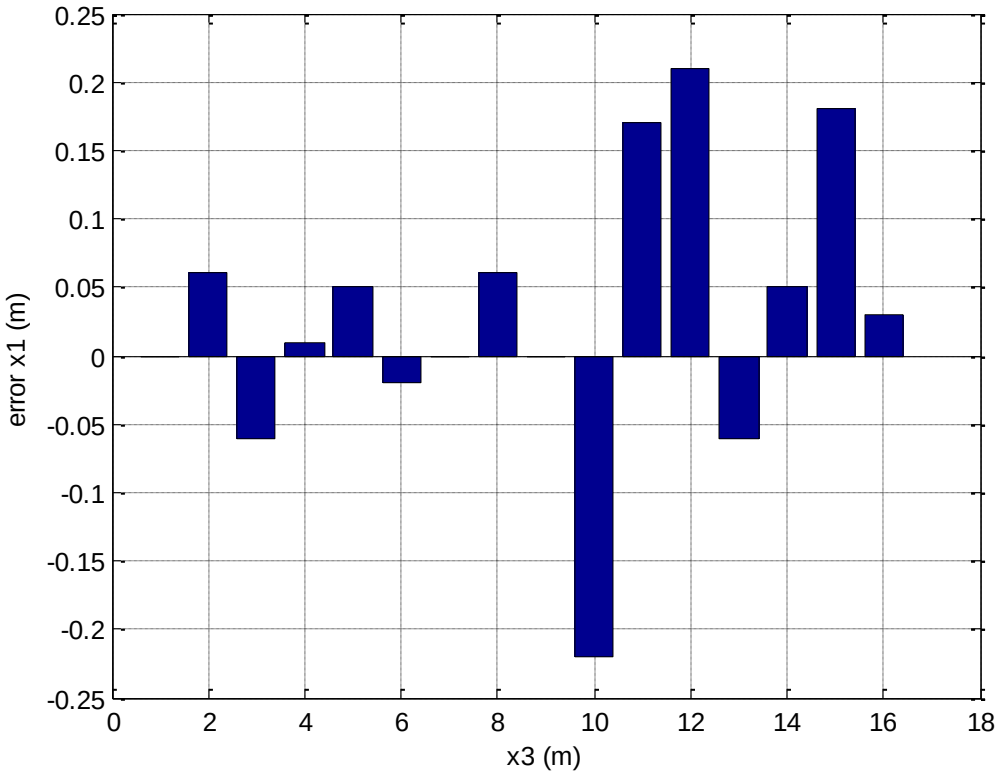

Figura 4.8. Error de medida de x1 en la segunda prueba

Si se compara con el resultado de la primera prueba, parece que incluso ha mejorado, aunque se podría achacar a una coincidencia estadística. Lo que nos interesa es el error cometido en *x3* para conocer si el ajuste ha mejorado la medida en cierto modo. En las figuras 4.8 y 4.9 se muestran los errores absoluto y relativo de la medida de *x3* en la segunda prueba.

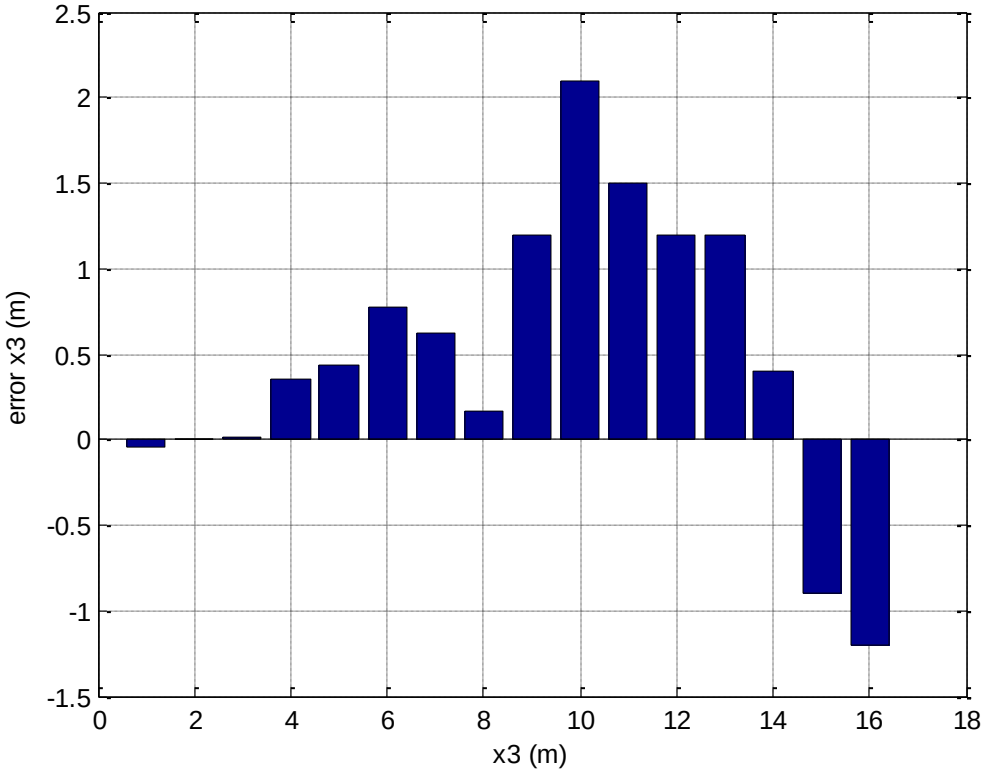

Figura 4.9. Error de medida de x3 en la segunda prueba

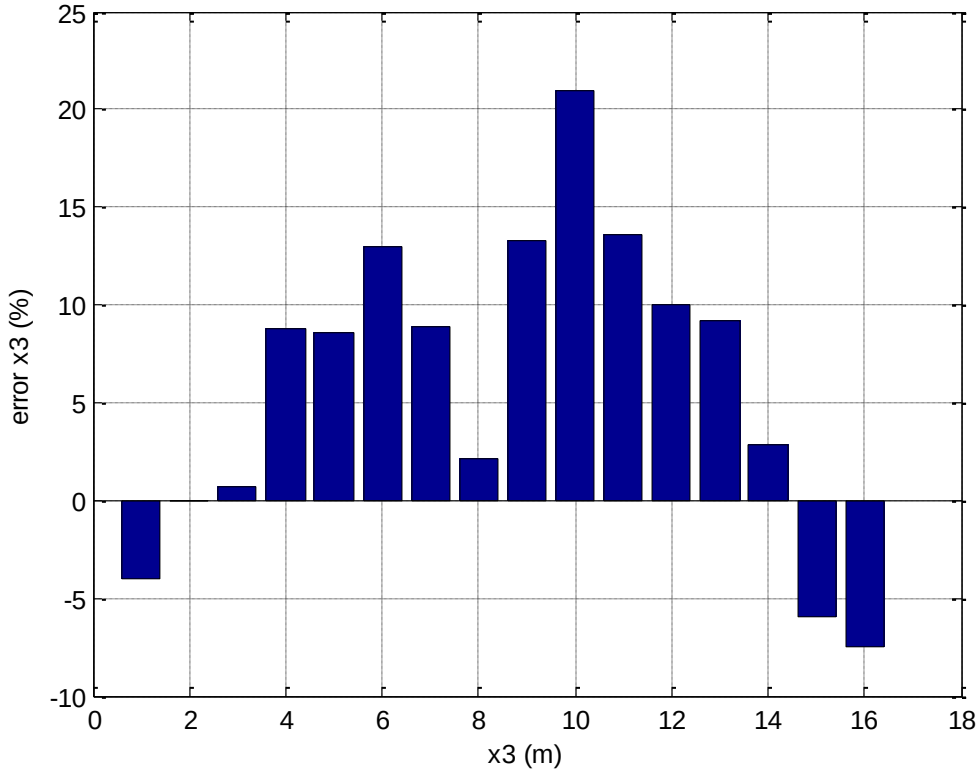

Figura 4.10. Error relativo de medida de x3 en la segunda prueba

Si se compara con la primera prueba, la mejora es evidente: el error no supera el metro hasta los nueve metros, comparado con los cinco de la medida sin calibrar. Además, el error relativo está acotado por debajo del 15% en 15 de las 16 muestras frente a siete de 15 anteriormente. Por último, antes de los diez metros, el error es inferior al 10% en siete de las nueve medidas.

Interesa saber si el tipo de cono influye en el error. Para ello, se ha hecho un análisis de varianza o ANOVA (Analysis of Variance) del error relativo de *x3* según el tipo de cono. El análisis ANOVA es una herramienta matemática que permite cuantificar la dependencia del resultado de un experimento respecto a uno o más parámetros. En nuestro caso, queremos saber si el error de *x3* depende del tipo de cono. Se ha escogido el error relativo para mitigar la dependencia del error con la distancia. Matlab nos ha devuelto, entre otras cosas, las medias según el tipo de cono, que son $(0.1196\ 0.0368\ 0.0325\ 0.0475)$; y un valor p de 0.4. El alto error medio de los conos azules puede atribuirse a la medida para $x3 = 10\text{m}$, donde se comete un error inusualmente alto. Si se escoge una probabilidad umbral típica del 5%, con los datos que tenemos no se puede descartar la hipótesis nula, esto es, que el error no depende del tipo de cono.

Por último, se ha hecho otra prueba en la que aparecen dos conos en pantalla intentando simular posiciones reales de pista: tramo recto, curva y línea de meta. Los resultados se muestran en la tabla 4.4:

Tipos	x1 reales (m)	x1 medidas (m)	x3 reales (m)	x3 medidas (m)	Comentarios
Azul y amarillo	-1.5, 1.5	-1.37, 1.37	5, 5	5.5, 5.5	
Azul y amarillo	-1.5, 0.25	-1.5, 0.26	6, 9	6.7, 9.4	
Naranjas chico y grande	-1.5, -1.5	-1.29, -1.09	5, 6	5.24, 5.12	Se ha confundido el cono grande con uno chico

Tabla 4.4. Resultados de la prueba con varios conos

El sistema funciona relativamente bien, salvo en el último caso, donde ha clasificado mal uno de los conos. Esto puede deberse a una falta de entrenamiento junto con el azar.

Hasta ahora se ha estudiado la precisión del sistema, pero queda por ver otro aspecto importante: el rendimiento. Por cada medida de posición hecha durante las pruebas, se ha guardado una instantánea de la imagen mostrada por el programa. En dicha captura aparecen los FPS instantáneos y mínimos. Si se repasan dichas capturas, los FPS mínimos registrados en las dos pruebas son 28 FPS. Sin embargo, en todas las capturas los FPS actuales se encuentran en el intervalo entre 48 y 70 FPS. Si se permiten caídas puntuales de rendimiento, sería posible implementar el sistema en tiempo real en una máquina menos potente. Para estudiar esta posibilidad, convendría añadir la posibilidad de registrar los FPS en todo momento y poder trabajar con ellos en Matlab, por ejemplo. De este modo, se podría estimar mejor la potencia computacional necesaria para un porcentaje de FPS mínimo. Por poner una referencia, las redes neuronales suelen funcionar con precisión de 32 bits de coma flotante (FP32) y la GTX 1050 usada aquí tiene una potencia de 1.86TFLOPS de FP32 [6]. Es posible que solo sean necesarios 1.5 o incluso 1.3 TFLOPS de potencia para el tiempo real, dependiendo de la distribución de FPS deseada.

Capítulo 5. Consideraciones para su futura integración

En este breve capítulo se ofrecerán ciertos consejos para la integración de las funcionalidades de la librería dentro de un programa mayor que se encargue del control de un vehículo Formula Student. Las sugerencias son las siguientes:

1. Colocar la cámara en el frontal del vehículo, centrada y a unos 20-25cm del suelo. De esta forma, tiene un campo de visión amplio, se puede referenciar el resto del vehículo respecto a la cámara (considerando ésta el origen de coordenadas del sistema) y minimiza el error de detección de bordes debido a la perspectiva.
2. Suponiendo que el programa final tiene información precisa de la velocidad del vehículo, tanto módulo como dirección, es buena idea investigar y considerar implementar un filtro de Kalman a la estimación de posición de los conos. Éste es un tipo especial de filtro que combina la entrada de un sensor (nuestro algoritmo) con una predicción de la medida (variación de la posición según la velocidad del vehículo) con el fin de lograr un error inferior al del sensor o la predicción solos. openCV implementa funciones de Kalman que podrían facilitar su futura implementación.
3. Con la misma suposición que antes, es recomendable guardar un registro de los conos detectados para tener el mapeo de la pista tras una vuelta. Dicho registro se puede dividir en cuatro arrays, uno por cada tipo de cono. Se deben mantener actualizadas las posiciones de los conos y la posición del coche respecto a los conos.
4. Se deben establecer reglas para la inclusión de un cono nuevo al registro. Puede darse el caso de un breve falso positivo (lo que podría denominarse un cono fantasma), un cono mal clasificado o, algo más

frecuente, que un cono detectado anteriormente desaparezca durante unos cuantos fotogramas, un falso negativo. Una posible estrategia es que, al detectar un posible candidato a cono en un fotograma, éste ha de reaparecer en el siguiente a cierta distancia según la velocidad del vehículo. Si se cumple esta condición durante un número determinado de fotogramas, se añade el cono al registro como definitivo.

5. Opcionalmente, si se desea algo más de rendimiento, se puede paralelizar el proceso. En el código de ejemplo *object_detection.cpp* de openCV, dependiendo de la versión de C++ utilizada se divide el trabajo en varios hilos de proceso, montando un esquema similar a un *pipeline*. Con esto se consigue que varias fases del programa se ejecuten al mismo tiempo que la red neuronal, ahorrando tiempo. El principal problema es la dificultad añadida a la programación de todo el sistema.
6. Es recomendable usar una cámara de buena calidad. No hablamos de una con una alta resolución o tasa de refresco, pues un flujo de video a 720p y 30 FPS ya supera las necesidades de nuestro sistema. Hablamos de una cámara que presente una imagen clara, con poco ruido y poca distorsión del color. Todos estos fenómenos afectan a la detección.
7. Respecto a la cámara, también es interesante conocer de antemano su campo de visión, determinado por su longitud focal. Una cámara con una longitud focal alta tiene menos campo de visión, por lo que perderá de vista los conos antes y puede tener problemas en curvas muy cerradas. Por contrapartida, los objetos lejanos son representados por una mayor cantidad de píxeles, permitiendo potencialmente la detección a mayores distancias.
8. Una estrategia que usan muchos equipos de Formula Student es, en las pruebas, realizar un primer recorrido lento para mapear bien el circuito y recorrerlo a máxima velocidad en el resto de pruebas, ya que el resultado se juzga por la vuelta más rápida, no por el total o la media. Se recomienda implementar dicha estrategia en el software de control del vehículo.
9. Si el desarrollo del vehículo autónomo va a necesitar uno o dos años, es recomendable prestar atención a la publicación de nuevas versiones de YOLO-tiny. Cuando se empezó este proyecto, se entrenó y utilizó la tercera versión por ser la que estaba disponible en aquel momento. Más adelante, se lanzó la cuarta versión y, aprovechando que había que reentrenar la red cono los conos de prueba, se usó esa nueva versión.

10. La mejor opción a la hora de escoger un sistema informático para instalar en el vehículo probablemente sea un mini PC orientado a juegos. Son pequeños, tienen alta capacidad de integración y la refrigeración sería fácil de adaptar. Por motivos económicos, se recomienda un modelo de hace cinco años (incluso de segunda mano si es posible) y que tenga fuente de alimentación externa. Así, no sería difícil obtener un equipo barato que solo necesite un cambio de voltaje para funcionar con el sistema eléctrico del vehículo.

Capítulo 6. Conclusiones y trabajo futuro

Se ha conseguido diseñar y probar un sistema de detección y posicionamiento de conos relativamente aceptable. Con una *webcam* barata y un PC con una tarjeta gráfica de gama media-baja de hace varias generaciones ha sido posible montar un programa capaz de detectar y clasificar correctamente los conos en casi cualquier situación, estimar su posición horizontal con gran precisión independientemente de la distancia y estimar su posición en profundidad con un margen de error razonable hasta los nueve metros. Y todo esto funcionando en tiempo real sin interrupciones excepto en el arranque del programa, durante las primeras décimas de segundo.

Con las limitaciones descubiertas, es necesario programar el futuro software de control de vehículo acordemente. Por ejemplo, en un tramo recto, el vehículo podrá posicionarse en el centro de la pista sin muchos problemas y podrá detectar curvas más adelante, pero no podrá conocer con precisión la distancia a la curva ni lo estrecha que es hasta que se haya acercado lo suficiente.

Un aspecto bastante desafortunado de este trabajo ha sido la imposibilidad de implementarlo sobre una FPGA. Se ha invertido, sin éxito, bastante esfuerzo en ello y, como consecuencia, se ha demorado varios meses la finalización del presente trabajo. Esto quizá podría haberse evitado de haberse hecho un pequeño estudio de viabilidad al inicio del proyecto.

Existe bastante margen de mejora, especialmente con la estimación de *x3*, como se ha visto. Entre las posibles mejoras se encuentran:

1. Reentrenar la red neuronal con más conos naranjas, tanto chico como grande. Como ya se dijo en el apartado de entrenamiento, buena parte de las imágenes del *set* se sacaron de competiciones Formula Student reales. El problema es que, debido a la naturaleza del propósito de cada tipo de cono, hay muchos más conos azules y amarillos que naranjas. Se

ha visto en las pruebas que, ocasionalmente, no se detectan o se clasifican mal dichos conos, especialmente cuando se encuentran a más de diez metros de la cámara. Añadir más conos naranjas al *dataset* debería ayudar a aliviar este problema.

2. Investigar y probar más algoritmos de estimación de *x3*. Este apartado es posiblemente tan amplio que podría considerarse un trabajo aparte. Para empezar, se podría probar la estimación utilizando solo la altura, el ancho u otra combinación de ambos distinta de la media. También se puede probar otro algoritmo que aprovecha que el cono se encuentra en el plano del suelo y la cámara está a una distancia h_0 conocida $x3 = -f\frac{h_0}{y2}$ (6.1)

   para $y2 < 0$. Además, existen muchas posibilidades para el ajuste de la medida: reducirlo a ajuste lineal, ajuste a trozos, ajuste no polinómico, ajuste según el tipo o el tamaño del cono, etc. Es imprescindible que cada ajuste sea probado y se estudie la distribución de su error para compararlo con los demás.

3. Se puede implementar multiproceso para mejorar en cierta medida el rendimiento. Puesto que el procesado de cada con es independiente en casi todas las fases del programa, muchos bucles *for* se pueden paralelizar. Se puede implementar el paralelismo con la librería *thread* de C++ o, quizá más sencillo, con la API openMP. Si se lleva a cabo, hay que tener en cuenta que se necesitan mecanismos de acceso, como *mutex*, a los vectores y a la imagen para evitar conflictos.
4. Implementar el sistema para que utilice openCL en lugar de CUDA y evaluar su impacto en el rendimiento. El uso de openCL permitirá el despliegue de la red nuronal y de la aplicación en GPU no fabricadas por Nvidia, tales como AMD o ARM.
5. Estudiar en el futuro la posibilidad de implementar este sistema en un MPSoc (Multi-Processor System on Chip) de Xilinx. Los MPSoC son chips que incluyen CPU y GPU de marca ARM más una parte de lógica programable o PL (Programmable Logic). En principio, con el uso de openCL su implementación sería posible, pues openCV soporta arquitectura ARM. En todo caso, habrá que tener en cuenta los costes de desarrollo y de adquisición de la placa, especialmente si es un modelo de automoción.

Aparte se encuentran las sugerencias ya expuestas en el capítulo anterior.

Apéndice A. *Repositorio del proyecto*

Se ha creado un repositorio accesible públicamente en Github a través del siguiente enlace: https://github.com/CPM1996/MART_CV. En él se encuentran disponibles:

- Una copia de la presente memoria.
- Los tres ficheros fuente utilizados (conos.hpp, conos.cpp y main.cpp).
- Un fichero CMakeLists.txt para compilar el proyecto usando CMake.
- Los ficheros .cfg y .weights que describen la red YOLOv4-conos-tiny.
- El fichero calibración.txt con la longitud focal y los parámetros de ajuste utilizados durante la segunda prueba.
- Una carpeta Entrenamiento con la estructura descrita en la figura 3.2, exceptuando el ejecutable *darknet* y los ficheros .cfg y .conv.29, que incluye el *set* de entrenamiento utilizado.
- Dos carpetas, Prueba_1 y Prueba_2 con las capturas del programa durante las pruebas.

Si se quiere entrenar una red neuronal con el *dataset* disponible, es necesario seguir las instrucciones de compilación y utilización de *darknet* descritas en [5]. Se recomienda no mover el ejecutable de la carpeta una vez compilado debido al uso de librerías dinámicas.

Para compilar el programa desarrollado, es necesario tener instalado las API openCV y Nvidia CUDA y CMake. Con ello, solo es necesario desplazarse a la carpeta de trabajo en una terminal de comandos y ejecutar los comandos

```
cmake .
```

```
make
```

Si la compilación ha tenido éxito, se puede utilizar el programa como se describió en el apartado 4.1.

Referencias

[1] "FSG Competition Handbook 2020", 2020, accesible en pdf en https://www.formulastudent.de/fileadmin/user_upload/all/2020/rules/FSG20_Competition_Handbook_v1.0.pdf

[2] F. Chollet, *Deep Learning with Python*, Shelter Island, Nueva York: Manning, 2018.

[3] A. Bochkovskiy, C. Wang y H. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection", Abril 2020, accessible en pdf en https://arxiv.org/pdf/2004.10934.pdf

[4] D. A. Forsyth y J. Ponce, *Computer Vision: A Modern Approach,* Upper Saddle River, Nueva Jersey: Prentice Hall, 2013

[5] A. Bochkovskiy, "YOLOv4 – neural networks for object detection", Agosto 2020, accessible en formato HTML en https://github.com/AlexeyAB/darknet

[6] "Nvidia Geforce GTX 1050 Specs", disponible en formato HTML en https://www.techpowerup.com/gpu-specs/geforce-gtx-1050.c2875