

**DETECCIÓN Y SEGUIMIENTO DE MÚLTIPLES OBJETOS EN TIEMPO REAL
PARA VEHÍCULOS AUTÓNOMOS.**



Res. No. 16740, 2017-2021.

Vigilada MinEducación.

**CAMILO ANDRÉS MOSQUERA VICTORIA
2146814
GERMAN ANDRES DUSSAN NARVAEZ
2146519**

**UNIVERSIDAD AUTÓNOMA DE OCCIDENTE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE AUTOMÁTICA Y ELECTRÓNICA
PROGRAMA INGENIERÍA MECATRÓNICA
SANTIAGO DE CALI
2020**

**DETECCIÓN Y SEGUIMIENTO DE MÚLTIPLES OBJETOS EN TIEMPO REAL
PARA VEHÍCULOS AUTÓNOMOS.**



Res. No. 16740, 2017-2021.

Vigilada MinEducación.

CAMILO ANDRÉS MOSQUERA VICTORIA

2146814

GERMAN ANDRES DUSSAN NARVAEZ

2146519

Proyecto de grado para optar al título de Ingeniero Mecatrónico

Director

JUAN CARLOS PERAFAN VILLOTA
INGENIERO ELÉCTRICO

Codirector

VICTOR ADOLFO ROMERO CANO
Ingeniero Mecatrónico

UNIVERSIDAD AUTÓNOMA DE OCCIDENTE

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE AUTOMÁTICA Y ELECTRÓNICA

PROGRAMA INGENIERÍA MECATRÓNICA

SANTIAGO DE CALI

2020

Nota de aceptación:

**Aprobado por el Comité de Grado en
cumplimiento de los requisitos exigidos
por la Universidad Autónoma de
Occidente para optar al título de
Ingeniero Mecatrónico**

Jesús Alfonso López
Jurado

José Luis Paniagua
Jurado

Santiago de Cali, 19 de marzo de 2020

AGRADECIMIENTOS

Los autores de este trabajo brindan su gratitud:

A Dios por darnos la vida y así mismo brindarnos el conocimiento y dedicación necesaria para solucionar los diferentes problemas presentados en el desarrollo de este trabajo

A nuestros padres, que durante todo el trayecto de la carrera nos apoyaron incondicionalmente, quienes nos hicieron las personas que somos en la actualidad. Gracias a ellos culminamos esta etapa de nuestras vidas y gracias a nuestros padres atrevemos a soñar en cosas mucho más grandes.

A Juan Carlos Perafan, nuestro director de tesis, quien nos apoyó y ayudó inmensamente, quien tuvo la paciencia no solo para indicarnos nuestras fallas, sino para ayudarnos a entender y corregirlas, quien nos guio de principio a fin y creyó en nuestras capacidades y nuestro proyecto.

A los profesores Walter mayor y Víctor romero, por los conocimientos aportados para el proyecto.

CONTENIDO	pág.
RESUMEN	13
INTRODUCCIÓN	15
1. OBJETIVOS	17
1.1 OBJETIVO GENERAL	17
1.2 OBJETIVOS ESPECÍFICOS	17
2. ANTECEDENTES	18
3. MARCO TEÓRICO	20
3.1 DATASET DE IMÁGENES DE VEHICULOS PARA MACHINE LEARNING	20
3.1.1 BIT vehicle dataset	20
3.1.2 Cityscapes Image Pairs	21
3.1.3 GTI Vehicle Image dataset	22
3.1.4 KITTI object detection with bounding boxes	22
3.1.5 Nepalese vehicles	23
3.1.6 Stanford Cars Dataset	24
3.1.7 COCO Dataset	24
3.2 REDES CONVOLUCIONALES	25
3.2.1 Ejemplos de redes convolucionales	26
3.2.1.1 MobileNet v1	26
3.2.1.2 MobileNet v2	27

3.2.1.3 Faster R-CNN	28
3.2.1.4 Inception v1	30
3.2.1.5 Inception v2	32
3.2.1.6 Inception v3	33
3.2.1.7 ResNet	34
3.2.1.8 YOLO (You Only Look Once)	35
3.3 FILTRO DE KALMAN	37
3.4 ALGORITMO HÚNGARO	38
3.5 MÉTODO DE EVALUACIÓN IOU	39
3.6 TRANSFER LEARNING	39
4. METODOLOGIA	41
4.1 SUBSISTEMA DE DETECCIÓN	42
4.1.1 Selección del dataset	43
4.1.2 Selección de la interfaz a trabajar	44
4.1.2.1 Tensorflow	44
4.1.2.2 Darknet	46
4.1.3 Selección de la plataforma para el reentreno	48
4.1.3.1 Tensorflow	48
4.1.3.2 Darknet	49
4.2 SUBSISTEMA DE SEGUIMIENTO	52
4.2.1 Procesamiento en tiempo real	52
4.2.2 Implementación del seguidor	53
4.3 SUBSISTEMA DE TOMA DE DECISIONES	56

4.4 ACOPLAMIENTO DEL SISTEMA A UNA TARJETA DE DESARROLLO JETSON TK1	62
5. CONCLUSIONES	71
BIBLIOGRAFÍA O REFERENCIAS	72
ANEXOS	79

LISTA DE FIG.S

	pág.
Fig. 1. Ejemplo del dataset BIT-vehicle [19].	21
Fig. 2. Ejemplo del dataset Cityscapes Images Pairs [20].	21
Fig. 3. Ejemplo del dataset GTI Vehicle Image [21].	22
Fig. 4. Ejemplo del dataset KITTI (2D) [22].	23
Fig. 5. Ejemplo del dataset Vehicles-Nepal [23].	23
Fig. 6. Ejemplo del Stanford Cars Dataset [24].	24
Fig. 7. Ejemplo del COCO dataset [25].	25
Fig. 8. Tres formas de convolución [28].	26
Fig. 9. Arquitectura de la red MobileNet [28].	27
Fig. 10. Vista general de la arquitectura de MobileNetV2 [29].	28
Fig. 11. RPN [30].	29
Fig. 12. Arquitectura de la red Faster R-CNN [30].	30
Fig. 13. Bloque "Inception" [31].	31
Fig. 14. Arquitectura de Inception v1 [31].	31
Fig. 15. Arquitectura general de la red InceptionV2 [32].	32
Fig. 16. Modulo "Inception" original vs Nuevo módulo "Inception" [32].	33
Fig. 17. Modulo "Inception" [32].	33
Fig. 18. Arquitectura de la red ResNet [33].	34
Fig. 19. Problema de degradación [33].	34
Fig. 20. Salto de conexiones [34].	35
Fig. 21. Arquitectura de YOLOv1 [35].	36
Fig. 22. Matriz del algoritmo húngaro [37].	38

Fig. 23. “Overlapping” de verdades y detecciones [38].	39
Fig. 24. Machine learning vs Transfer learning [39].	40
Fig. 25. Diagrama del proyecto.	41
Fig. 26. Subsistema de detección.	42
Fig. 27. Imagen del dataset de KITTI [22].	43
Fig. 28. Precisión vs Tiempo de inferencia [42].	46
Fig. 29. Subsistema de seguimiento.	52
Fig. 30. Ejemplo del algoritmo IOU [45].	54
Fig. 31. Subsistema de decisiones.	56
Fig. 32. Toma de puntos y resultado final.	57
Fig. 33. Ejemplo de selección de puntos [48].	58
Fig. 34. Resultado final del cambio de perspectiva.	58
Fig. 35. Resultado del algoritmo de medición de distancias y ángulos (t=0) [48].	61
Fig. 36. Resultado del algoritmo de medición de distancias y ángulos (t=1) [48].	61
Fig. 37. Resultado del algoritmo de medición de distancias y ángulos (t=2) [48].	62
Fig. 38. Uso de la aplicación Ip Webcam.	64
Fig. 39. Conexión a la plataforma web de Ip Webcam.	64
Fig. 40. Adaptador para el carro.	65
Fig. 41. Salida del carro.	65
Fig. 42. Salida del adaptador.	66
Fig. 43. Tarjeta embebida en funcionamiento.	66
Fig. 44. Seguimiento y medición de la distancia de un carro (t=0).	68

Fig. 45. Seguimiento y medición de la distancia de un carro (t=1).	68
Fig. 46. Seguimiento y medición de la distancia de un carro (t=2).	69
Fig. 47. Seguimiento y medición de la distancia de un carro (t=3).	69
Fig. 48. Seguimiento y medición de la distancia de un carro (t=4).	70

LISTA DE TABLAS

pág.

Tabla I.	45
Tabla II.	47
Tabla III.	48
Tabla IV.	49
Tabla V.	50
Tabla VI.	51
Tabla VII.	51
Tabla VIII.	55

LISTA DE ANEXOS

pág.

Anexo A. Validación del entrenamiento de las redes de Tensorflow con imágenes del dataset de KITTI	79
Anexo B. Comprobación de distancia.	89
Anexo C. Validación de detección y seguimiento.	91
Anexo D. Filtro de Kalman.	98
Anexo E. Validación del entrenamiento con YOLO con imágenes del dataset de KITTI.	100
Anexo F. Diagrama general del proyecto.	106
Anexo G. Diagrama del proyecto	107
Anexo H. Resultados de la evaluación de los detectores.	108

RESUMEN

Este documento presenta el desarrollo de un sistema de detección y seguimiento de múltiples objetos en tiempo real para vehículos autónomos. El primer paso, consistió en seleccionar un tipo de *dataset* que permitiera entrenar el sistema de una manera consistente a la realidad de la conducción humana; el tipo seleccionado fue el Dashcam, que consiste en videos tomados desde una cámara localizada en el parabrisas del auto, grabando continuamente la vista frontal de la carretera. El segundo paso, consistió en seleccionar dos técnicas, una para detección de vehículos y la otra para seguimiento de estos.

Dado los grandes avances que se han obtenido en la última década en sistemas basados en inteligencia artificial, se hizo uso de esta para crear el sistema de detección basado en el uso de una red convolucional *You Only Look Once* (YOLO) de la interfaz Darknet. Este sistema funciona de manera supervisada en su fase de entrenamiento y ha demostrado ser capaz de detectar una gran cantidad de diferentes clases de objetos.

Para nuestro caso particular, se realizó un reentrenamiento con el fin de que la detección se limitará solo a vehículos. En este orden de ideas, para realizar la detección, cada imagen procesada es dividida en $S \times S$ celdas, sobre las cuales se predicen N posibles cajas delimitadoras de objetos, a las cuales les es asignado una probabilidad basados en el grado de certidumbre relativo al objeto que se desea aprender a detectar. Después de obtener estas predicciones se procede a eliminar las cajas que estén por debajo de un umbral previamente definido. A las cajas restantes se les eliminan posibles objetos que fueron detectados por duplicado y se mantiene únicamente el más exacto de ellos.

Con el fin de garantizar una conducción segura, cada vehículo detectado debe ser posteriormente seguido mientras este se encuentre dentro del rango de actuación del vehículo autónomo. Dado que en un ambiente real se pueden tener varias detecciones de diferentes vehículos al mismo tiempo, el seguimiento de cada uno de ellos de manera precisa y eficiente fue llevado a cabo utilizando un estimador conocido como Filtro de Kalman, el cual fue replicado para crear un banco de estimadores, de manera tal que cada estimador dentro del banco sea asignado a un vehículo en particular detectado dentro de la escena.

La detección y el posterior seguimiento de cada vehículo dentro de la escena, por sí solas, no permiten al vehículo autónomo tomar decisiones acerca de frenado y reducción o aumento de velocidad. Para esto, se hace imprescindible calcular las

distancias relativas entre cada vehículo detectado y el vehículo autónomo, lo cual se logró utilizando una transformación de perspectiva.

Después de la selección de los algoritmos de detección y seguimiento, se prosiguió con la validación de estos mismos. Cabe resaltar que, para obtener buenos resultados, se analizaron previamente diferentes *dataset* los cuales debían contener, para nuestro caso específico, muestras de imágenes de diferentes clases de carros. Teniendo en cuenta lo mencionado, se utilizó una tarjeta JETSON NVIDIA TK1 que permitió compilar los algoritmos eficientemente gracias a la GPU integrada en la tarjeta embebida. Finalmente, esta tarjeta fue montada sobre un vehículo para realizar pruebas de detección, seguimiento y posterior toma de decisiones en cuanto a manejo reactivo en tiempo real.

INTRODUCCIÓN

Uno de los primeros problemas resueltos en el campo de la visión computacional fue la detección de objetos, la cual consiste en poder separar el elemento a ser estudiado del resto de la imagen (comúnmente conocido como *background*). Esta detección se extendió luego a videos, teniendo en cuenta que un video puede ser visto como una secuencia de imágenes, conocidas como marcos (*frames*) o cuadros.

Una problemática nueva surge al intentar no solo detectar un objeto sobre cada marco, sino también seguirlo. El seguimiento (*tracking*) en visión computacional es el proceso de localizar uno o múltiples objetos en movimiento sobre una secuencia de imágenes capturadas por una video cámara. El seguimiento de un objeto es el proceso que consiste en estimar la trayectoria de dicho elemento en una imagen mientras este se encuentra en movimiento. En otras palabras, es un método en el que se sigue un objeto a través de una secuencia de imágenes para determinar el movimiento relativo respecto a otros objetos.

El problema se acrecienta en ambientes donde la detección ocurre sobre múltiples objetos. Para estos casos, se requiere de una metodología más elaborada que consiste en tres pasos: detección, predicción y asociación de datos. Con la detección se localiza el objeto de interés en un marco; la predicción usa esta primera detección y la dinámica asociada al movimiento de este para predecir la localización de ese objeto en el siguiente marco; dado que puede darse el hecho en el cual dos o más objetos detectados en un marco n tengan una misma predicción de localización en el marco $n+1$, se hace necesario aplicar el paso de asociación con el fin de eliminar esta ambigüedad.

Ahora bien, en el seguimiento por detección, un gran reto del MOT en tiempo real es lograr asociar robustamente objetos sobre imágenes ruidosas detectados en un nuevo marco de video con los objetos seguidos previamente.

De acuerdo a un reporte del 2016 de World Economic Forum [1], para el 2040 el número de carros en las carreteras se doblara, alcanzando la marca de los 2 billones, y de acuerdo a un reporte de Business Insider, en el 2020 habrían más de 10 millones de carros autónomos (con al menos una funcionalidad que define a un carro autónomo, como acelerar, frenar o maniobrar el volante con muy poca o ninguna intervención humana) andando por las calles, cifra que se doblará para el 2025, de los cuales muchos funcionaran sin la intervención de seres humanos [2]. Esto ha generado la necesidad de mejorar la movilidad urbana, lo que ha llevado a

grandes avances en tecnologías autónomas. Por esta razón muchas empresas están apostando a los carros autónomos como el futuro de los sistemas de transporte. La Sociedad de Ingenieros de Automoción (SAE International) define los niveles de autonomía de un carro autónomo de la siguiente forma: El nivel 0 es un carro que no tiene ningún tipo de automatización; el nivel 1 es un carro que puede realizar pequeñas tareas como hacer pequeñas maniobras o acelerar con muy poca o ninguna intervención del conductor; en el nivel 2 se encuentra el control de crucero adaptativo y el sistema de autopiloto, donde el carro puede realizar maniobras de seguridad, pero el conductor debe seguir alerta al volante; el nivel 3 aun requiere un conductor humano, pero bajo ciertas condiciones de tráfico o del ambiente el conductor puede derivar funciones críticas de seguridad a el carro; en el nivel 4 el carro puede conducirse casi por completo de forma autónoma, pero aun requiere de un conductor humano para alguna situación muy específica; finalmente en el nivel 5 el carro es completamente autónomo y no requiere intervención humana.

Empresas como Waymo, General Motors y Argo IO han logrado un nivel 4 de autonomía, pero el nivel más alto que se encuentra en el mercado actualmente es el 2, con los vehículos de Tesla. En Colombia el nivel de autonomía que se maneja en las carreteras es el 1, con carros de marca Chevrolet y Volvo [3], además actualmente hay unos cuantos carros Tesla con nivel de autonomía 2, pero sus funciones de autonomía están restringidas a países europeos.

Buscando un mayor nivel de autonomía para los vehículos en Colombia se propone el diseño de un sistema que permite detectar, seguir y determinar la distancia entre el vehículo autónomo y los vehículos a su alrededor y de esta manera poder tomar acciones rápidas ante la aproximación de otros vehículos. Para ello se implementó un sistema que consta de 3 subsistemas: el sistema de detección, en el cual se usa la red convolucional Tiny-YOLOv3 de la interfaz Darknet para realizar la detección de carros; el sistema de seguimiento, en el que se usan bancos de filtros de Kalman para realizar el seguimiento de los carros, y el algoritmo Húngaro y el método de evaluación de intersección sobre la union (IOU) para realizar la asignación de las detecciones a los filtros; y el subsistema de toma de decisiones, en el cual se usa el triángulo de similitud y una función trigonométrica para hallar la distancia y el Angulo de las detecciones con respecto a el ego-car, respectivamente. Todo el sistema fue seleccionado pensando en poderse implementar en una tarjeta embebida móvil, la NVIDIA Jetson TK1, y poder probar el sistema en tiempo real en un carro. Con base a lo anterior surge la siguiente pregunta problema: ¿Es posible implementar un sistema MOT en tiempo real basado en el estado del arte actual de vehículos autónomos tipo 5?

1. OBJETIVOS

1.1 OBJETIVO GENERAL

Desarrollar un sistema de detección y seguimiento de múltiples objetos basados en el estado del arte actual de la detección de objetos para vehículos autónomos de tipo 5.

1.2 OBJETIVOS ESPECÍFICOS

- Seleccionar un dataset tipo Dashcam que permitan estudiar el seguimiento de vehículos.
- Seleccionar la técnica de detección y seguimiento de objetos a utilizar
- Implementar el algoritmo que permita realizar la detección de carros.
- Implementar el algoritmo que permita realizar el seguimiento de carros.
- Validar los algoritmos implementados en tiempo real.

2. ANTECEDENTES

La visión artificial tiene como objetivo generar descripciones inteligentes y útiles de escenas y secuencias visuales, así como de los objetos que aparecen en ellas, mediante la realización de operaciones sobre imágenes y videos. Estas operaciones se basan en la detección y el seguimiento de objetos, mejorando durante el tiempo la eficacia y la precisión de dichas operaciones. Con el objetivo de estudiar la detección y el seguimiento, existen trabajos cuyo objetivo es mejorar cada uno de los procedimientos que estos requieren. Asimismo, recientes trabajos de seguimiento de múltiples objetos han demostrado avances significativos, los cuales se expondrán a continuación:

En 2017, Agarwal y Suryavanshi [4] realizaron un trabajo en el cual construyeron un MOT usando redes neuronales profundas, en el que combinaron el detector Faster R-CNN (*Regions with convolutional neural networks*) [5] con la arquitectura GOTURN (*Generic object tracking using regression networks*) [6]. Ellos dividieron el MOT en dos categorías: la detección de múltiples objetos y la correspondencia de esos objetos.

Para la primera parte usaron la CNN (*convolutional neural network*), que enmarca la región de interés, y la combinaron con una CNN basada en regiones, la cual detecta la presencia del objeto en estas. Debido a que la red no requiere conocimiento de la clase del objeto para realizar la detección, la arquitectura es flexible y permite ser ajustada a múltiples diseños. Para el seguimiento de objetos utilizaron el GOTURN, que usa imágenes del tiempo “t” y “t-1” de un video en tiempo real y las ingresa individualmente en diferentes CNN. La salida de estas CNN es ingresada a otra red neuronal que trata de hacer la relación de correspondencia del objeto, buscando características similares en el objeto original. Un inconveniente que se evidencia en el proceso es que la arquitectura GOTURN solo tiene la capacidad de realizar seguimiento a un solo objeto, por lo cual se necesita de la arquitectura CNN para poder realizarlo a múltiples objetos.

Por otra parte, en 2018, Beaupré, Bilodeau y Saunier [7] presentaron un trabajo para mejorar el seguimiento de múltiples objetos. Se centraron en cómo crear mejores imágenes de primer plano, las cuales luego pasan a ser la entrada de los trackers MKCF (*Multiple Kernelized Correlation Filter Tracker*) [8] y el UT (*Urban tracker*) [9]; estos dos ayudaron a estudiar los efectos de dichas mejoras en el rendimiento o efectividad de los trackers. Para mejorar las imágenes de primer plano aplicaron los procesos *Canny Edge detector* [10], *optical Flow* [11] y *Background subtraction* [12]; con el primero localizaron las regiones de interés, con el segundo separaron objetos, mientras que con el tercero obtuvieron los bordes de los objetos en primer plano,

con los que se ajustaron el tamaño de los objetos; una vez reunida esta información, los tres procesos generaban una nueva imagen binaria.

Asimismo, en 2019, Li, Dobler, Song, Feng y Wang [13] presentaron un trabajo bastante innovador; la detección y el tracking no se hacen de forma separada, es decir que no se emplea un detector y un tracker por aparte, sino que crean una red de estructura unificada para realizar simultáneamente la tarea de MOT. Esta red toma un GoP (Grupo de imágenes) consecutivos como una representación 3D, y detecta los objetos que se están moviendo dentro del GoP por medio de “tubos”, utilizan una red convolucional 3D [14] y una VGG (*Visual geometry group*) [15] con convoluciones en 2D como redes base para realizar la extracción de características espaciales y temporales en los videos de entrada, también un *spatial transformer module* [16] es usado para lidiar con la variación de orientación de los objetos para que posteriormente una TPN (*tube proposal network*) [17] genere “tubos” para los objetos detectados. Por último, una etapa post-TPN refina la clasificación y los parámetros de localización de los “tubos” localizados para una detección más acertada.

3. MARCO TEÓRICO

La detección y seguimiento de múltiples objetos en vehículos autónomos es una forma de incrementar la seguridad de cada una de las personas que están en el entorno, por esta razón a través de cámaras y un sistema inteligente, se detectarán múltiples objetos en movimiento, para que el vehículo autónomo pueda decidir, teniendo en cuenta las imágenes captadas y así poder determinar el movimiento más viable o la decisión más acertada.

3.1 DATASET DE IMÁGENES DE VEHICULOS PARA MACHINE LEARNING

Para el entrenamiento de una red es necesario tener en cuenta un *dataset* que contenga imágenes que permitan una debida detección; entre mayor cantidad de muestras de imágenes, mayor será la eficacia del entrenamiento. Al trabajar con la perspectiva del ego-car (es decir que solo veremos lo que se encuentra enfrente del carro) se deben buscar *dataset* que contengan este tipo de imágenes; además las detecciones deben ser en su mayoría vehículos en carreteras, por lo que las detecciones en perspectiva superior no serían de gran utilidad.

Entre los *dataset* que se destacan para el desarrollo de vehículos autónomos se mencionan los siguientes:

3.1.1 BIT vehicle dataset

Originado en el laboratorio inteligente de información de tecnología de Beijín, este *dataset* incluye 9,850 imágenes de vehículos. Las imágenes se dividen en las siguientes categorías: bus, microbús, minivan, sedan, SUV y camión. El paper correspondiente al *dataset* se encuentra en [18]. En la Fig. 1 se muestran varios ejemplos de detecciones en el *dataset*.

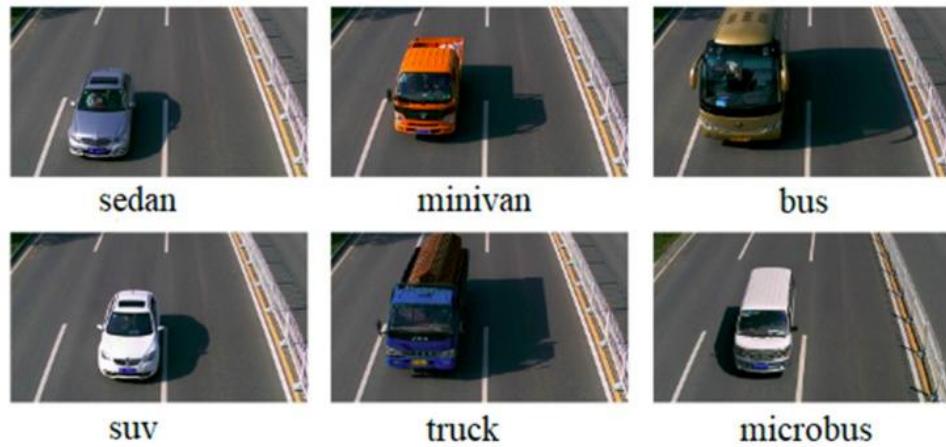


Fig. 1. Ejemplo del dataset BIT-vehicle [19].

Si bien el *dataset* es grande y tiene clases de detecciones que se ajustan bastante bien al proyecto, la perspectiva no es la adecuada ya que la cámara parece estar enfocada en tomar la parte delantera de los vehículos, pero en ambientes reales nos encontraremos con casi todas las perspectivas de estos.

3.1.2 Cityscapes Image Pairs

Contiene videos etiquetados que fueron tomados de vehículos en Alemania. Este *dataset* contiene 2,975 imágenes de entrenamiento y 500 imágenes de validación. Cada archivo de imagen es de 256x512 pixeles y cada archivo está compuesto con la foto original en la parte media de la izquierda de la imagen, junto a la imagen etiquetada en la parte media de la derecha. El paper correspondiente al dataset se encuentra en [20]. En la Fig. 2 se muestra un ejemplo de las imágenes del *dataset*.



Fig. 2. Ejemplo del dataset Cityscapes Images Pairs [20].

Este es un ejemplo de un *dataset* en donde la perspectiva de la cámara se adapta bien al proyecto ya que la parte derecha de las imágenes puede ser eliminada fácilmente. El problema es que el *dataset* no tiene etiquetas de detecciones, pues está diseñado para proyectos de segmentación, por lo que no es realmente útil para el trabajo.

3.1.3 GTI Vehicle Image dataset

Este *dataset* contiene 3,425 imágenes de ángulo trasero de vehículos en carretera, así como 3,900 imágenes de carreteras ausentes de cualquier vehículo. El paper correspondiente al *dataset* se encuentra en [21]. En la Fig. 3 se muestra un ejemplo de las imágenes del *dataset*.



Fig. 3. Ejemplo del dataset GTI Vehicle Image [21].

Este *dataset* se enfoca solo en una perspectiva de las detecciones, y aunque esa perspectiva (trasera) es útil para el proyecto, solo una parte es necesaria para el aprendizaje del detector.

3.1.4 KITTI object detection with bounding boxes

El conjunto de datos contiene 7481 muestras, para ser más específicos tiene 6347 para el entrenamiento, 711 para la prueba y 423 para la validación; todo esto anotado con cuadros delimitadores 3D. KITTI contiene un conjunto de tareas de

visión creadas con una plataforma de conducción autónoma. El punto de referencia completo contiene muchas tareas, como estéreo, flujo óptico, odometría visual. El paper correspondiente al dataset se encuentra en [22]. En la Fig. 4 se muestra un ejemplo de las imágenes del *dataset*.



Fig. 4. Ejemplo del dataset KITTI (2D) [22].

Este *dataset* presenta detecciones en múltiples perspectivas; visto desde la parte frontal del carro, la altura de la cámara es ideal, y las detecciones están en carreteras, por lo que se ajusta perfectamente al proyecto.

3.1.5 Nepalese vehicles

Consiste en un total de 30 videos de tráfico tomados en la calle de Katmandú (Nepal). Este conjunto de datos contiene 4,800 imágenes recortadas de los videos. De las 4,800 imágenes, 1,811 son vehículos de dos ruedas y 2,989 de 4 ruedas. El paper correspondiente al dataset se encuentra en [23]. En la Fig. 5 se muestra un ejemplo de las imágenes del *dataset*.



Fig. 5. Ejemplo del dataset Vehicles-Nepal [23].

Este *dataset* fue tomado desde cámaras de tráfico, por lo que no es una perspectiva que se ajuste al proyecto.

3.1.6 Stanford Cars Dataset

Este conjunto de datos contiene 16,185 imágenes de 196 diferentes clases de carros. Los datos están divididos en 8,144 imágenes de entrenamiento y 8,041 imágenes de prueba, donde cada clase ha sido dividida en dos partes iguales. El paper correspondiente al dataset se encuentra en [24]. En la Fig. 6 se muestra un ejemplo de las imágenes del *dataset*.



Fig. 6. Ejemplo del Stanford Cars Dataset [24].

Este *dataset* tiene una perspectiva aceptable de las detecciones, pero las clases son muy específicas, y lo que se busca es una idea general de lo que es un carro.

3.1.7 COCO Dataset

Es un conjunto de datos de detección, segmentación y subtitulación de objetos a gran escala. Contiene varias características como lo es la segmentación de objetos, reconocimiento en contexto, segmentación de super píxeles, 1,5 millones de instancias de objetos, 80 categorías de objetos, 91 categorías de objetos y 250,000 personas con puntos clave. El paper correspondiente al dataset se encuentra en [25]. En la Fig. 7 se muestra un ejemplo de las imágenes del *dataset*.



Fig. 7. Ejemplo del COCO dataset [25].

Este es otro *dataset* que se ajusta al proyecto muy bien. El problema radica en la amplitud de este, las clases son muy diversas y solo se usarían unas cuantas para realizar el entrenamiento.

3.2 REDES CONVOLUCIONALES

Las redes convolucionales (CNN) son un algoritmo de *deep learning* que toma una imagen de entrada y le asigna importancia a diferentes aspectos y características de la imagen. Su arquitectura es similar a los patrones de conectividad de las neuronas del cerebro humano, inspirada por la corteza visual.

Una CNN es capaz de capturar las dependencias espaciotemporales en una imagen a través de la aplicación de filtros. Este tipo de arquitectura (CNN) se ajusta mejor a *dataset* con imágenes debido a que reduce el número de parámetros involucrados y a que los pesos de las neuronas son reusables.

Para una mayor profundización sobre las redes convolucionales se tiene una referencia [26], donde se especifica el funcionamiento y los procesos que realizan dichas redes.

3.2.1 Ejemplos de redes convolucionales

En la actualidad nos encontramos con un gran número de CNN, siendo su principal uso el analizar imágenes. Sus aplicaciones varían desde reconocimiento en imágenes y video, clasificación de imágenes, análisis de imágenes médicas hasta procesamiento de lenguaje natural. A continuación, daremos una breve introducción a las redes consideradas en el proyecto.

3.2.1.1 MobileNet v1

Las redes MobileNet fueron presentadas como modelos eficientes dirigidos a aplicaciones con dispositivos embebidos y dispositivos móviles. El paper correspondiente a la red se encuentra en [27]. Fueron construidas con el concepto de convoluciones profundas separables (depth wise separable convolutions). Para ilustrar esto mejor se presenta la Fig. 8.

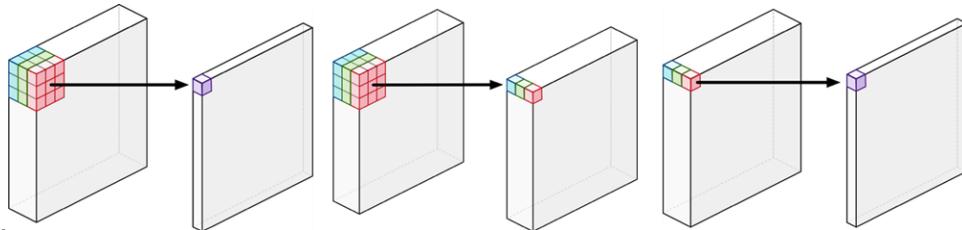


Fig. 8. Tres formas de convolución [28].

A la derecha se tiene una convolución típica, en la que la salida de los tres canales se junta en un solo punto, en este tipo de convoluciones se realizan productos punto con el mismo filtro (con cambios en su tamaño) sobre la imagen; en el medio se tiene la convolución separable profunda, donde los tres canales tienen su propia salida de convolución, en este tipo de convolución las operaciones se realizan individualmente sobre cada canal, lo que hace que cada canal tenga sus propios pesos; al final se tiene una convolución de un punto, allí el Kernel es un solo punto con profundidad igual a los canales de la imagen, opera sobre toda la imagen, y su salida se junta en un solo punto como en una típica convolución. La principal ventaja de la convolución separable profunda es su eficacia computacional, ya que este tipo

de convoluciones requieren muchas menos operaciones matemáticas que las convoluciones 2D.

Como se aprecia en la Fig. 9 las arquitecturas MobileNet combinan tanto la convolución separable como la convolución de un punto.

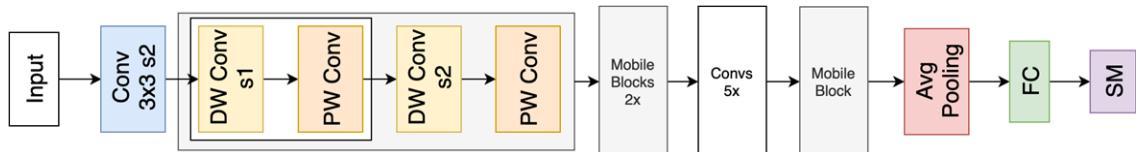


Fig. 9. Arquitectura de la red MobileNet [28].

La arquitectura consta de una capa convolucional, seguida de 3 bloques “Mobile”, los cuales constan de una capa de convolución separable profunda (DW), seguida de una capa de convolución de punto (PW), luego otra capa DW y una nueva capa PW (en la Fig. 9 se muestra un bloque “Mobile” expandido, y otros 2 retraídos). Luego de estos bloques siguen 5 más de convolución (los cuales constan de una capa DW seguida de una capa PW), conectados a una capa “Mobile”, antes de finalizar con capas de agrupamiento y de clasificación.

Gracias a la reducción de operaciones por la implementación de las convoluciones profundas separables, MobileNet (con 4 millones de parámetros) es 3 veces más rápida que la red Inception (la cual tiene 24 millones de parámetros), y es 10 veces más rápida que la red VGGNet-16 (que tiene 138 millones de parámetros); además es capaz de correr a 20 FPS en un smartphone, con una precisión del 70.6%.

3.2.1.2 MobileNet v2

MobileNet V2 introdujo cambios en las conexiones residuales y en las capas *bottleneck* (las conexiones residuales ayudan a los gradientes a penetrar mucho más la red). El paper correspondiente a la red se encuentra en [29]. En la Fig. 10 se muestra la arquitectura general de la red.

Input	Operator	<i>t</i>	<i>c</i>	<i>n</i>	<i>s</i>
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	-

Fig. 10. Vista general de la arquitectura de MobileNetV2 [29].

Mientras la secuencia de las convoluciones de punto (PW) y profundas (DW) siguen presentes, el propósito de la convolución de punto (PW) cambió: primero, comprime el número de canales al proyectar puntos de datos con dimensiones grandes a un tensor de menor dimensión. Esta reducción de información que se adentra a la red es llamada “bottleneck”, por lo que la capa de proyección es normalmente llamada “bottleneck layer”.

Segundo, la capa de expansión PW expande el número de canales antes de que la información sea enviada a una convolución DW.

Entonces cuando una imagen entra al bloque PW, primero se expande en términos de canales, luego se aplica una DW, y finalmente se comprime la información en un número menor de canales. Es una idea parecida a lo que hace la red Inception.

En términos de eficacia, la MobileNet v2 con 3.5 millones de parámetros trabaja mucho mejor que su predecesora.

3.2.1.3 Faster R-CNN

El paper correspondiente a la red se encuentra en [30]. Esta red se encarga de reemplazar el algoritmo de búsqueda selectiva lenta con una red neuronal más rápida, específicamente se introdujo la red de propuesta de región (RPN, Fig. 11).

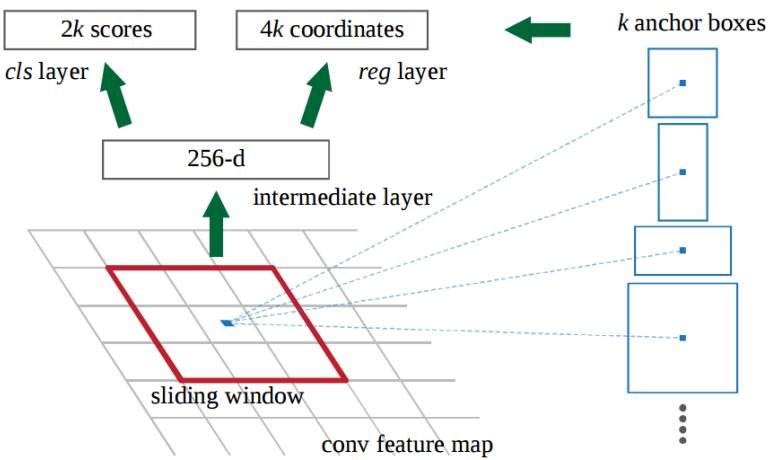


Fig. 11. RPN [30].

El RPN funciona de la siguiente forma:

- En la última capa de una red neuronal convolucional, una ventana deslizante se mueve a través de un mapa característico y mapea en una dimensión menor.
- Para cada localización de la ventana deslizante, se genera múltiples posibles regiones basadas en k , cajas de anclaje de relación fija.
- Cada región propuesta consiste en un resultado de objetividad para la región y 4 coordenadas representando la caja delimitadora de la misma.

Los puntajes de $2k$ representan la probabilidad de *SoftMax* de cada una de las cajas delimitadoras, siendo k un objeto. Cabe destacar que las coordenadas de salidas de las cajas delimitadoras no clasifican ningún objeto, su trabajo sigue siendo el proponer objetos en las regiones. Una vez se tenga las regiones propuestas, se añade una capa de agrupamiento, algunas capas totalmente conectadas y finalmente una capa clasificadora *SoftMax* y una caja delimitadora como regresor. En pocas palabras, Faster R-CNN = RPN + Fast R-CNN. En la Fig. 12 se muestra la arquitectura red Faster R-CNN.

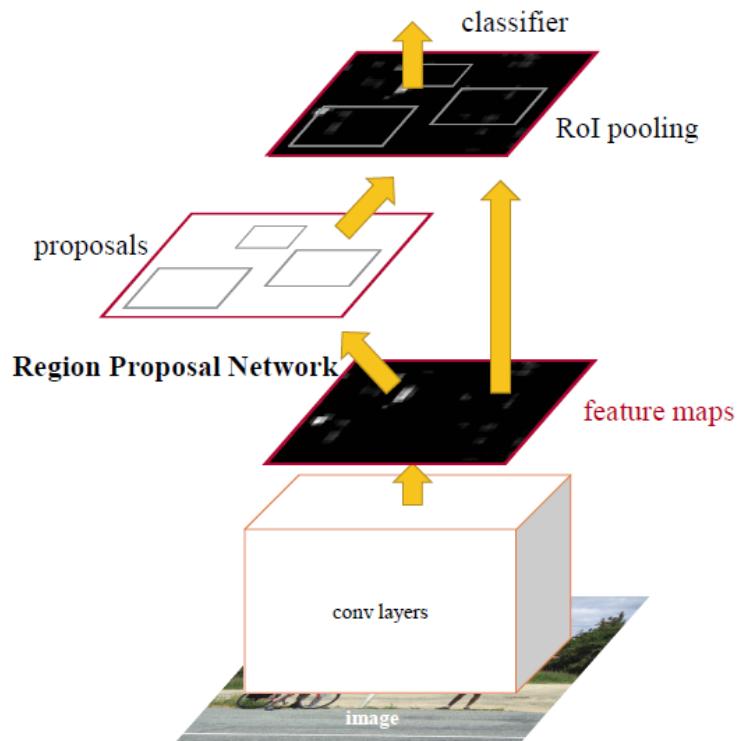


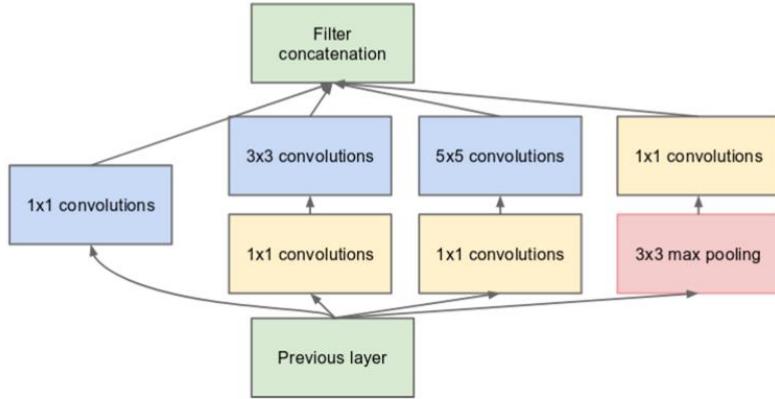
Fig. 12. Arquitectura de la red Faster R-CNN [30].

Al tener en cuenta RPN Y Fast R-CNN, la red adquiere una mayor velocidad y precisión para la detección de objetos.

3.2.1.4 Inception v1

El paper correspondiente a la red se encuentra en [31]. La red Inception fue una introducción importante en el desarrollo de las redes CNN clasificadoras. Antes de Inception, las redes CNN más populares solo apilaban una capa tras otra.

La idea de la primera Inception, V1, fue implementar filtros de múltiples tamaños que operaran en el mismo nivel. Esto volvería la red más amplia, en vez de más profunda. En la Fig. 13 se muestra el módulo “Inception”.



(b) Inception module with dimension reductions

Fig. 13. Bloque "Inception" [31].

Las convoluciones 1x1 son usadas para reducir el número de canales, resultando en un incremento tanto en profundidad como en amplitud.

En total tiene 9 módulos “Inception” apilados, 22 capas de profundidad y usa el agrupamiento global promedio al final del ultimo módulo “Inception”. En la Fig. 14 se muestra la arquitectura de la Inception V1.

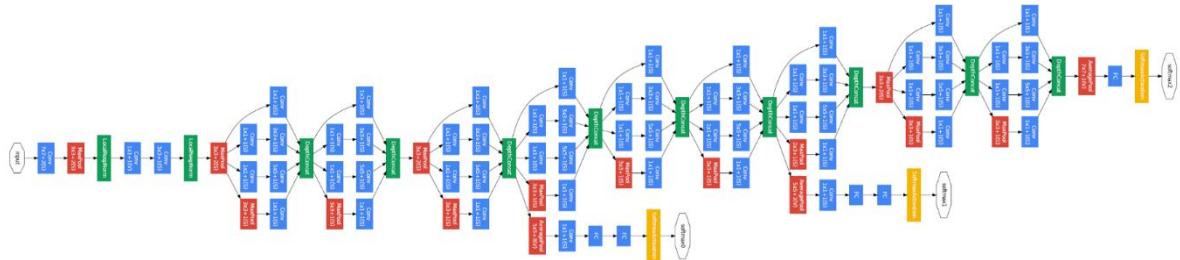


Fig. 14. Arquitectura de Inception v1 [31].

Las cajas púrpuras son clasificadores auxiliares, mientras que las partes amplias son módulos “Inception”.

Los módulos auxiliares fueron introducidos para solucionar el problema de la desaparición del gradiente (la mitad de la red “muere”). La idea es aplicar SoftMax a la salida de 2 módulos “Inception” y computar la pérdida auxiliar. La función de pérdida total es la suma de la pérdida auxiliar y la pérdida real (donde a la primera

se le da un peso de 0.3 en la suma). Esta pérdida auxiliar solo es tenida en cuenta en el momento del entrenamiento, en el momento de la inferencia es ignorada.

3.2.1.5 Inception v2

La motivación tras las redes Inception V2 e Inception V3 (las cuales salieron al mismo tiempo, siendo Inception V3 prácticamente igual a Inception V2, ya que esta fue usada para realizar pruebas y ajustes y V3 es la versión con estos ajustes) es evitar la pérdida de información al realizar reducciones de dimensión (*representational bottleneck*) aplicando métodos de factorización. El paper de la red Inception-V2 e Inception-V3 se encuentra en [32].

Lo que se hizo fue factorizar una convolución 5x5 a 2 convoluciones de 3x3 para mejorar la velocidad. En la Fig. 15 se muestra la arquitectura general de la red InceptionV2.

type	patch size/stride or remarks	input size
conv	$3 \times 3 / 2$	$299 \times 299 \times 3$
conv	$3 \times 3 / 1$	$149 \times 149 \times 32$
conv padded	$3 \times 3 / 1$	$147 \times 147 \times 32$
pool	$3 \times 3 / 2$	$147 \times 147 \times 64$
conv	$3 \times 3 / 1$	$73 \times 73 \times 64$
conv	$3 \times 3 / 2$	$71 \times 71 \times 80$
conv	$3 \times 3 / 1$	$35 \times 35 \times 192$
$3 \times$ Inception	As in figure 5	$35 \times 35 \times 288$
$5 \times$ Inception	As in figure 6	$17 \times 17 \times 768$
$2 \times$ Inception	As in figure 7	$8 \times 8 \times 1280$
pool	8×8	$8 \times 8 \times 2048$
linear	logits	$1 \times 1 \times 2048$
softmax	classifier	$1 \times 1 \times 1000$

Fig. 15. Arquitectura general de la red InceptionV2 [32].

En la Fig. 16 se puede comparar la diferencia entre el módulo “Inception” de la arquitectura original y la arquitectura V2.

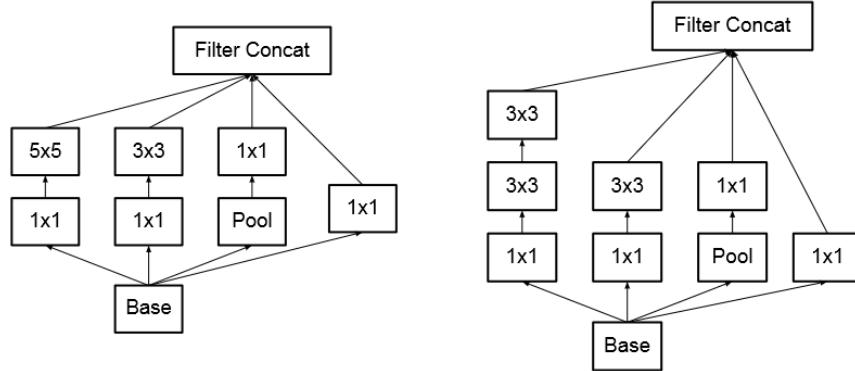


Fig. 16. Modulo “Inception” original vs Nuevo módulo “Inception” [32].

Luego esto se llevó al extremo, al factorizar convoluciones $n \times n$ a una combinación de $1 \times n$ y $n \times 1$, como se ilustra en la Fig. 17. Este método probó ser 33% menos costoso computacionalmente que realizar una convolución 3×3 .

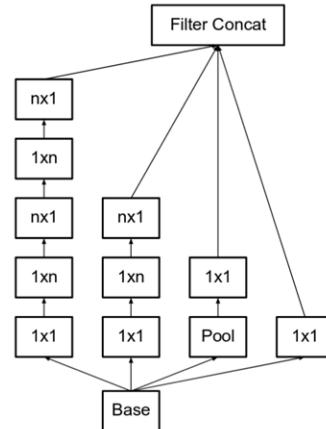


Fig. 17. Modulo “Inception” [32].

Finalmente se expandieron los filtros para reducir la pérdida de información; en vez de quedar más profundos se hicieron más amplios.

3.2.1.6 Inception v3

La versión 3 de la red incorporó un optimizador RMSProp, así como la factorización de operaciones convolucionales 7×7 , la implementación del BatchNorm en los

clasificadores auxiliares y un componente de regularización (*label smoothing*), el cual previene que la red se vuelva demasiado segura.

La normalización de lotes (BatchNorm) fue introducida como la solución al cambio covariante interno (*Internal covariate shift*). Esta sucede gracias a que la distribución de la entrada de las capas cambia durante el entrenamiento, debido a que el parámetro de las capas anteriores también cambia. Este fenómeno (*Internal covariate shift*). BatchNorm obliga a la entrada de las capas a tener una distribución parecida en cada *step* del entrenamiento al normalizar la media y la varianza de las características en cada nivel de representación.

3.2.1.7 ResNet

El paper correspondiente a la red se encuentra en [33]. Es una abreviación de “*Residual Network*” (red residual), salió durante el LSVRC2012 (un concurso de clasificación). En la Fig. 18 se presenta la arquitectura de esta red.

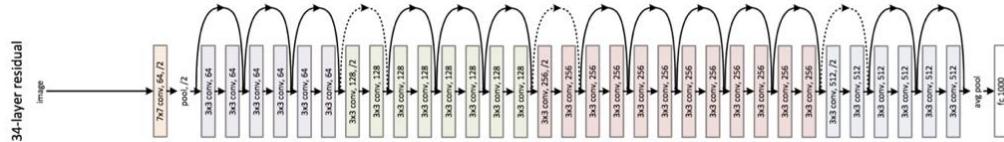


Fig. 18. Arquitectura de la red ResNet [33].

La idea detrás de la ResNet es solucionar el problema de degradación, que ocurre cuando se trata de entrenar una red muy profunda. Este problema tiene 2 fases, vistas en la Fig. 19.

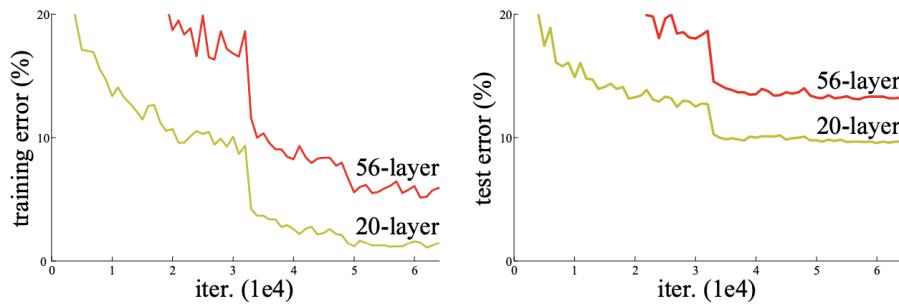


Fig. 19. Problema de degradación [33].

En la Fig. 19 se observa primero la saturación de la precisión (la red deja de aprender), y segundo la degradación de esta. El fuerte de la ResNet es el concepto de salto de conexiones, como se observa en la Fig. 20.

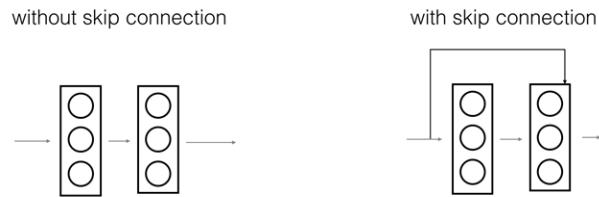


Fig. 20. Salto de conexiones [34].

En la izquierda se tiene una red apilando convoluciones, mientras que a la derecha se puede observar una red que también apila convoluciones, pero además añade la entrada original a la salida del bloque de convolución. Esto elimina el problema de la degradación al permitir al gradiente fluir por otro camino, además de que se asegura que las capas de alto nivel funcionen igual de bien que las capas de bajo nivel.

3.2.1.8 YOLO (You Only Look Once)

El paper correspondiente a la red se encuentra en [35]. YOLO es una red neuronal convolucional diseñada para la detección de objetos de la cual existen 3 versiones. Antes de la implementación de YOLO, los clasificadores de imágenes trabajaban escaneando toda la imagen para localizar el objeto. Este proceso de escanear la imagen completamente comienza con una “ventana” predefinida que produce un resultado booleano verdadero o falso, dependiendo de si el objeto específico se encuentra o no en la “ventana”. Luego de recorrer toda la imagen, esta “ventana” es agrandada y se vuelve a escanear toda la imagen de nuevo. Este método es llamado “*Sliding window*” (ventanas deslizantes). Además, redes como la R-CNN y Fast R-CNN usan el método de RPN (*Region Proposal Network*) para generar potenciales cajas de detecciones (*bounding boxes*), procesar dichas cajas con el clasificador, perfeccionar los bordes de estas cajas y eliminar duplicados de detecciones

Es necesario tener presente que YOLO es diferente de los métodos anteriormente mencionados, ya que trata el problema de detección de imágenes como un problema de regresión en vez de uno de clasificación, y corre una sola red neuronal convolucional para realizar todas sus tareas. Esto tiene beneficios tanto en

velocidad (el proceso de convolución se realiza una sola vez en toda la imagen para realizar las predicciones), como en la reducción de errores de fondo (ya que mira la imagen de forma entera en vez de por partes y puede sacar conclusiones globales, lo que disminuye los errores de falsas predicciones). Aun así, redes como la Faster R-CNN son más precisas, ya que el costo de la velocidad de inferencia viene con una reducción de precisión (especialmente con objetos pequeños o un grupo de objetos pequeños).

Por lo tanto, YOLO está diseñada como una red convolucional que está conformada por 24 capas convolucionales y 2 capas “Fully connected”. Las primeras 20 capas convolucionales son seguidas por una capa de agrupamiento medio, y las capas “Fully connected” son pre-entrenadas con el *dataset* de ImageNet (que cuenta con 1000 clases).

Las ultimas 4 capas convolucionales seguidas por las capas “Fully connected” son añadidas para entrenar la red para la detección de objetos, así la última capa predice la probabilidad de la clase y las cajas de detección. En la Fig. 21 se puede observar la arquitectura de YOLOv1.

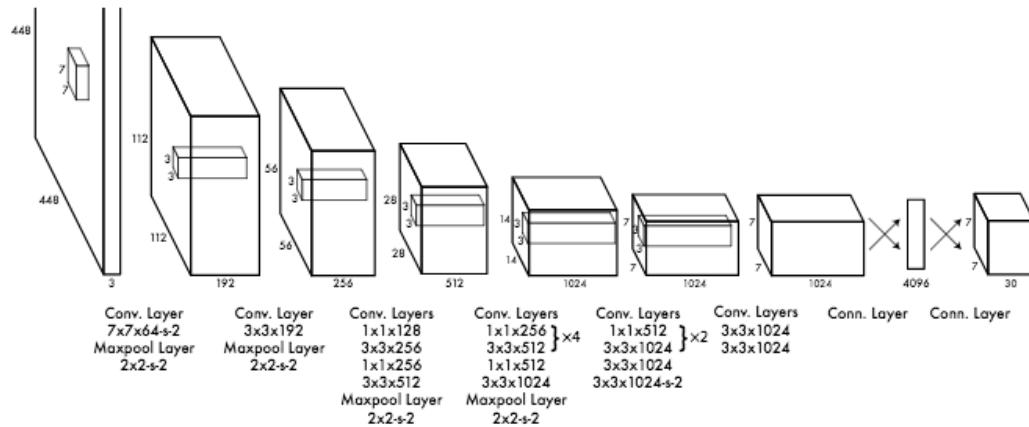


Fig. 21. Arquitectura de YOLOv1 [35].

La última capa usa una activación lineal, mientras que otras redes usan activaciones ReLU.

El algoritmo funciona dividiendo la imagen en celdas y por cada celda se predicen cajas de detección y su puntuación de confianza, además de la probabilidad de la

clase. La confianza es dada en términos de IOU (*Intersection over union*), que es básicamente mirar qué tanto una detección se intercala con el objeto verdadero (la forma en que la red aprendió a representar este objeto).

Ahora bien, YOLOv2 respecto a su predecesor, mejora el hecho de que la red no era muy buena detectando objetos muy cercanos y tendía a cometer errores de localización. Principalmente introduce cajas predeterminadas, y ahora la red pasa de predecir cajas de detecciones a predecir el offset de estas cajas. Además, introduce el uso de características granulares más finas, lo que logró que la predicción de objetos pequeños fuera mejor. Mientras que YOLOv3 añade pequeños detalles respecto a su predecesor, como el que las cajas de detecciones son predichas a diferentes escalas.

3.3 FILTRO DE KALMAN

El filtro de Kalman [36] es usado para estimar el estado de un sistema lineal, donde el estado se asume que está distribuido por una función gaussiana. El filtro de Kalman estima un proceso usando una forma de control de retroalimentación. El filtro evalúa el estado del proceso en cierto tiempo y luego obtiene la retroalimentación en forma de medidas de ruido. Las ecuaciones del filtro de Kalman están constituidas en dos grupos: las ecuaciones de tiempo actualizado y las ecuaciones de actualización de medición. Las ecuaciones de actualización de tiempo son las responsables de proyectar hacia delante (del tiempo) el estado actual y estimación de la covarianza del error para obtener el a priori estimado para el siguiente paso de tiempo. La ecuación de actualización de medición es responsable de la retroalimentación, que es usada para incorporar nuevas medidas en las estimaciones a priori, para obtener una mejor estimación.

A continuación, se presentan las ecuaciones del filtro de Kalman:

$$\bar{x}_k = Ax_{k-1} + Bu_k \quad (1)$$

$$\bar{P}_k = A\bar{P}_{k-1}A^T + Q \quad (2)$$

$$K_k = \bar{P}_k H^T (H\bar{P}_k H^T + R)^{-1} \quad (3)$$

$$x_k = \bar{x}_k + K_k(z_k - H\bar{x}_k) \quad (4)$$

$$P_k = \bar{P}_k (H K_k - I) \quad (5)$$

Donde (1) es la predicción del estado actual, (2) es el cálculo del error de la covarianza de la predicción, (3) es el cálculo la ganancia de Kalman, (4) es el cálculo del estado real y (5) es el cálculo de la covarianza del error real.

3.4 ALGORITMO HÚNGARO

El algoritmo húngaro es un algoritmo usado para resolver problemas de asignación. Tiene 2 componentes: los trabajos y las maquinas. Todas las maquinas pueden ser asignadas a cualquier tarea, pero cada máquina tiene un costo diferente para cada tarea. Lo que se requiere es asignar cada máquina a una tarea y cada tarea a una máquina de forma que el costo total de las asignaciones es minimizado. En la Fig. 22 se muestra un ejemplo de la matriz que contiene los trabajadores, las tareas y los costos de cada tarea respecto a los trabajadores.

	Task A	Task B	Task C
Worker 1	250	450	350
Worker 2	400	400	350
Worker 3	200	500	250

Fig. 22. Matriz del algoritmo húngaro [37].

3.5 MÉTODO DE EVALUACIÓN IOU

Es una métrica de evaluación usada para medir la precisión de un detector de objetos frente a un dataset particular. Usualmente se ve esta métrica de evaluación en retos de evaluación de objetos como el popular PASCAL VOC challenge. Para poder aplicar este método es necesario tener las etiquetas que especifican las coordenadas de los objetos en las imágenes y las coordenadas predichas por el detector.

El algoritmo premia las predicciones que mejor superpongan las verdades absolutas. En la Fig. 23 se muestran diferentes “overlapping” y los puntajes que el IOU les asigna.



Fig. 23. “Overlapping” de verdades y detecciones [38].

3.6 TRANSFER LEARNING

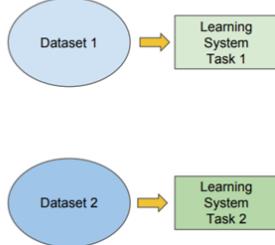
Es un método de machine learning donde un modelo diseñado para una tarea es reusado como el punto de partida para otra tarea, donde modelos pre-entrenados son usados como puntos de partida para tareas de visión computacional y procesamiento de lenguaje natural, dada la cantidad de recursos computacionales que se requieren para desarrollar un modelo de red neuronal.

Además, transfer learning solo funciona en *deep learning* si el modelo aprende características generales de su primera tarea. Esta forma de transfer learning es llamada *inductive transfer*, en la que el posible modelo es escogido basándose en

qué tan similar es la tarea que se quiere que aprenda. En la Fig. 24 se muestra una comparación entre machine learning tradicional y transfer learning.

Traditional ML vs Transfer Learning

- Isolated, single task learning:
 - Knowledge is not retained or accumulated. Learning is performed w.o. considering past learned knowledge in other tasks



- Learning of a new tasks relies on the previous learned tasks:
 - Learning process can be faster, more accurate and/or need less training data

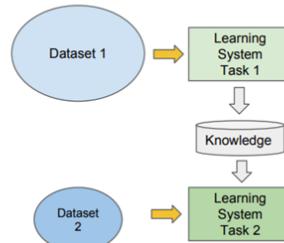


Fig. 24. Machine learning vs Transfer learning [39].

Transfer learning aprovecha el conocimiento ya adquirido por la red para aprender una nueva tarea, mientras que en machine learning el aprendizaje es aislado, el conocimiento no es retenido. Al realizar un reentreno de la red (o “*transfer learning*”) lo que se busca es disminuir la cantidad de pesos que se reentrenaran, usando los pesos y “*bias*” de la red ya pre-entrenada y recalculando solo los pesos de las conexiones de la última capa.

Entre los beneficios de realizar *transfer learning* se encuentra que el tiempo de entrenamiento es mucho menor que si se entrenara una red desde cero (teniendo en cuenta que por lo general las redes convolucionales tienen millones de neuronas), y que una red pre-entrenada necesitará menos datos de entrada para obtener resultados decentes.

4. METODOLOGIA

La Fig. 25 muestra el sistema propuesto para la detección y seguimiento de vehículos en tiempo real. Este a su vez puede dividirse en tres subsistemas principales: el subsistema de detección, el subsistema de seguimiento o *tracking* y el subsistema de toma de decisiones.

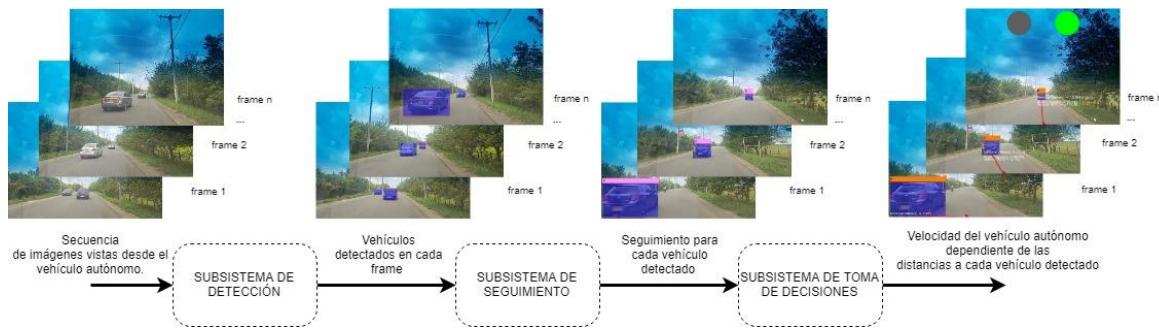


Fig. 25. Diagrama del proyecto.

En el Anexo F se puede observar la Fig. 25 de forma más amplia.

Para el desarrollo del subsistema de detección se reentrenaron 5 arquitecturas de redes neuronales convolucionales (específicamente MobileNet V2, Faster RCNN Inception V2, Faster RCNN ResNet, YOLOv3 y Tiny-YOLOv3) con el fin de detectar únicamente carros. Las dos tareas principales para el desarrollo adecuado de este subsistema fueron: la selección del *dataset* y la selección de la interfaz.

El subsistema de seguimiento tuvo como fundamento matemático la teoría probabilística y más específicamente el Filtro de Kalman, el cual permite estimar variables de estado no observables a partir de variables observables. Para nuestro caso, la variable a ser estimada es la posición de un automóvil específico en diferentes instantes de tiempo, con el fin de conocer la trayectoria trazada por este mientras se encuentra en el campo visual del vehículo autónomo. No obstante, hay que resaltar que la naturaleza en la detección de posibles autos que estén dentro del campo visual del vehículo autónomo se da a cada instante de tiempo, sumado al hecho de que un filtro de Kalman solo puede estimar una sola trayectoria al tiempo, hacen necesaria la posibilidad de crear un banco de este tipo de filtros, con el fin de obtener las trayectorias para múltiples vehículos.

El subsistema de toma de decisiones se encarga de definir distancias entre el vehículo autónomo y cada uno de los otros autos, siendo detectados en cada instante de tiempo. Esto permite tomar decisiones acerca de la velocidad que deberá tomar el vehículo autónomo con el fin de evitar colisiones.

El proyecto empieza por la recolección de los datos (Video) que se envían a la tarjeta embebida por medio de la aplicación IP Webcam [40]. OpenCV captura el video y envía las *frames*, una por una a ser procesadas, pasando primero por el detector que obtiene las coordenadas de los carros que detecte, luego estas detecciones son evaluadas para ver si existían en un *frame* anterior o si es la primera vez que aparecen en escena y con base a esto se guardan en una instancia de filtro de Kalman, el cual realizará el seguimiento del vehículo detectado mientras este permanezca en el rango de visión; finalmente se calcula la distancia y el ángulo para determinar qué tan cerca está el ego-car de los vehículos presentes.

A continuación, cada uno de los subsistemas mencionados previamente será expuesto en detalle.

4.1 SUBSISTEMA DE DETECCIÓN

En la Fig. 26 se muestra cómo se compone el subsistema de detección.

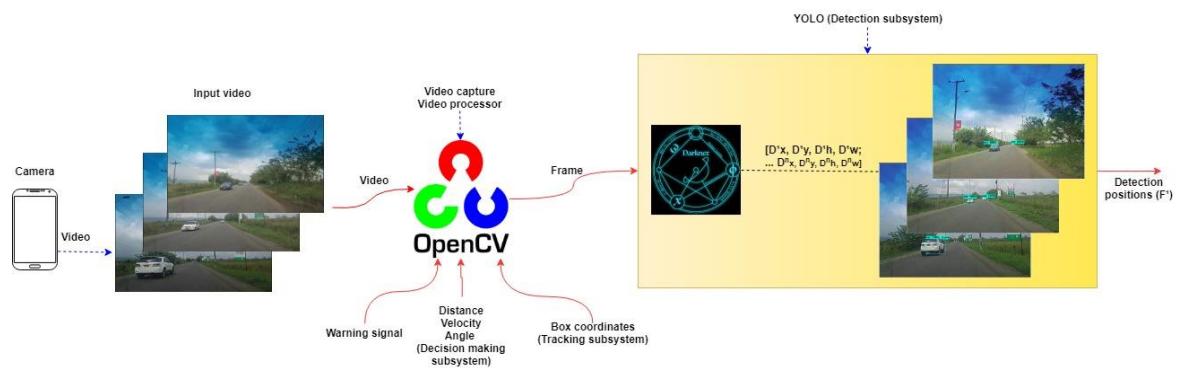


Fig. 26. Subsistema de detección.

4.1.1 Selección del dataset

Para la selección del *dataset* se tuvo en cuenta que este tuviera la mayor cantidad de imágenes de vehículos para que fuera mucho más eficaz el entrenamiento de la red, para que así se pudiera lograr un mejor desarrollo en el proyecto. Durante la investigación se encontraron muy buenos *dataset* como lo es, *Rain and Snow Traffic surveillance*, el cual consiste en un total de 22 videos cada uno con una duración de cinco minutos. Estos videos fueron capturados utilizando una cámara RGB y una cámara infrarroja térmica. Toda la información incluye cerca de 130,000 pares de imágenes térmicas RGB. También se encontró un *dataset*, el cual contiene una gran variedad de imágenes, este es **Stanford cars dataset**, este *dataset* de carros contiene gran entrenamiento y sets de pruebas para formar modelos que pueden identificar diferentes tipos de carros. El *dataset* de carros contiene 16,185 imágenes de 196 clases de carros. La información se divide entre 8,144 imágenes de entrenamiento y 8,041 imágenes de prueba, donde cada clase ha sido dividida en 50-50.

Cabe destacar que *Stanford cars dataset* y *Rain and Snow Traffic surveillance* era una buena opción, pero necesitábamos un mejor resultado en el entrenamiento de la red, por esta razón se optó por utilizar el *dataset* de KITTI ya que este se basa en el manejo autónomo de los carros, que también se especializa en la visión estéreo, flujo óptico, la odometría visual, y la detección y seguimientos de objetos 3D.

Este *dataset* contiene un conjunto de datos de detección de objetos, incluyendo imágenes monoculares y cuadros delimitadores. El conjunto de datos contiene 7481 muestras, para ser más específicos tiene, 6347 para el entrenamiento, 711 para la prueba y 423 para la validación, todo esto anotado con cuadros delimitadores 3D. En la Fig. 27 se observa una imagen tomada del *dataset*.



Fig. 27. Imagen del dataset de KITTI [22].

También una alternativa viable fue COCO dataset, ya que es un conjunto de datos de detección, segmentación y subtitulación de objetos a gran escala, la división de prueba no contiene anotaciones (solo imágenes). Las anotaciones panópticas definen 200 clases, pero solo utiliza 133.

4.1.2 Selección de la interfaz a trabajar

4.1.2.1 Tensorflow

Para la tarea de detección de objectos, Tensorflow cuenta con una variedad de redes convolucionales, caracterizadas por la cantidad de capas, la velocidad de detección, precisión y el *dataset* con el que se hayan entrenado; debido a que entrenar una red neuronal de forma completa es computacionalmente costoso, se elige reentrenar parcialmente una red pre-entrenada.

Teniendo en cuenta que para realizar un exitoso reentreno, la red para esta tarea debe haber aprendido previamente características generales de lo que deseamos reconocer, es necesario usar redes pre-entrenadas con vehículos, por lo que redes entrenadas con *dataset* como COCO, KITTI o IMAGENET son adecuadas para empezar el transfer learning.

Se escogieron 3 redes: La MobileNet V2 y la Faster RCNN Inception V2 (pre-entrenadas con el dataset de COCO), y la Faster RCNN ResNet (pre-entrenada con el *dataset* de KITTI); además el reentreno se realizará con el dataset de KITTI, especializado para carros autónomos.

En la Tabla I se muestra la velocidad de procesamiento y la precisión de detección (evaluado en las métricas del *dataset* usado para el pre-entreno) de las redes escogidas.

Tabla I.

Velocidad de procesamiento y precisión de detecciones de las redes de Tensorflow.

Red	Velocidad (ms)	Precisión promedio (mAP)
SSD MobileNet V2	31	22
Faster RCNN Inception V2	58	28
Faster RCNN ResNet	79	87

Estos datos fueron sacados del repositorio de Tensorflow Models [41], en donde los autores midieron la velocidad y precisión de las redes en una computadora con GPU Nvidia GeForce GTX TITAN X de 12 GB de memoria dedicada.

Antes de pensar en el reentreno, es necesario convertir el *dataset* a TFRecords, un formato de almacenamiento binario de Tensorflow que provee diferentes scripts con los cuales se pueden transformar varios *dataset* populares a este formato de almacenamiento, y además provee un script para transformar *dataset* creados por el propio usuario para facilitar el uso de su interfaz. El script usado fue *create_kitti_tf_record.py*, y para usarlo se especificó la carpeta donde se encontraba el *dataset*, la carpeta en la que se quiere guardar los TFRecords, las clases de vehículos requeridas, y el mapa de etiquetas del *dataset*.

Una vez se obtienen los TFRecords se procede a realizar el entrenamiento, una vez más Tensorflow Models provee 2 scripts para realizar los reéntrenos: *model_main.py* y *train.py*, con la diferencia que el primero es el más reciente y realiza el entreno y la evaluación de forma conjunta, mientras que el segundo solo realiza el entreno, y si se quiere realizar la evaluación se debe utilizar un script adicional.

Para usar *model_main.py* fue necesario especificar la dirección del archivo de configuración de la red, la carpeta donde se guardarán los progresos del reentreno, el número de *steps* que hará el reentreno y la cantidad de *steps* a los que se realizará una evaluación.

4.1.2.2 Darknet

Darknet en su página web nos presenta de entrada un gráfico (Fig. 28) en el cual compara el performance de sus redes frente a la de otras redes populares, usando el método de evaluación del *dataset* COCO, en el grafico se muestra la precisión vs el tiempo de inferencia de las redes, y se ve como la red YOLOv3 es capaz de lograr una precisión mayor que las demás redes en un tiempo de inferencia mucho más pequeño (tiempo de inferencia hace referencia al tiempo que le toma a la red para hacer sus predicciones).

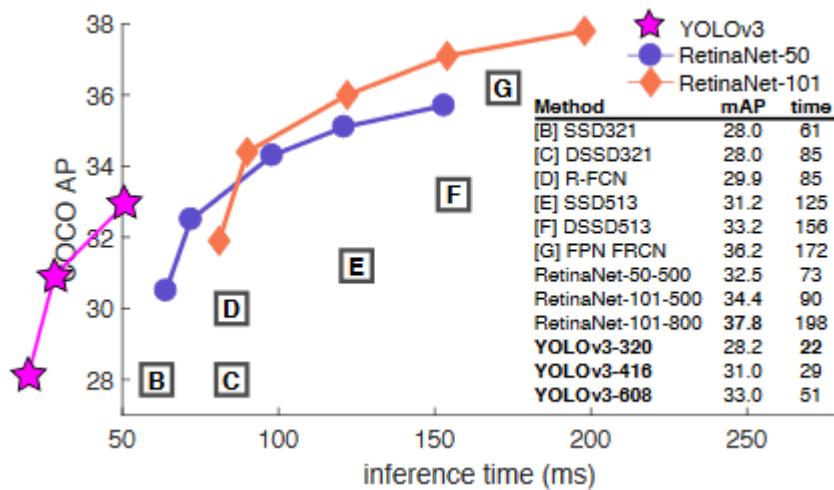


Fig. 28. Precisión vs Tiempo de inferencia [42].

De entre las 7 redes YOLO que tiene Darknet (YOLOv2, Tiny YOLO, YOLOv3-320, YOLOv3-416, YOLOv3-608, YOLOv3-tiny y YOLOv3-spp) se escogen las más recientes, YOLOv3 y YOLOv3-tiny para hacer el reentreno. En la Tabla II se presenta una comparación de la precisión y los FPS de las redes YOLO.

Tabla II.

Comparación de los FPS vs Precisión de las redes YOLO.

Red	Fps	Precisión promedio (mAP)
YOLOv2	40	48.1
Tiny YOLO	244	23.7
YOLOv3-320	45	51.5
YOLOv3-416	35	55.3
YOLOv3-608	20	57.9
YOLOv3-tiny	220	33.1
YOLOv3-spp	20	60.6

Estos datos fueron sacados del repositorio de Darknet [43]. Podemos observar que entre más crece el tamaño de la red, mayor es la precisión, pero este aumento de tamaño disminuye los FPS que se obtuvieron. Para lograr estos resultados las pruebas de FPS y precisión fueron realizadas en una GPU Nvidia GeForce Pascal Titan X de 12GB de memoria dedicada.

Teniendo en cuenta esto se decidió usar la versión pequeña de YOLOv3 (YOLOv3-tiny) así como también la versión completa (YOLOv3) para comparar resultados.

Antes de pensar en el reentreno, es necesario preparar los datos con los que se va a entrenar la red, ya que Darknet tiene su propio formato de etiquetas para las detecciones (*el cual provee información sobre la clase de detección y la ubicación*) por lo que es necesario convertir el *dataset* de KITTI al formato de Darknet.

Luego de obtener las nuevas etiquetas se procede a reentrenar la red con el comando *train* de Darknet, dejando que se itere hasta los 10000 *steps* (cada 1000 *steps* Darknet guarda los pesos actuales de la red, por lo que al final obtendremos 10 redes con diferentes valores de pesos). Finalmente se prueba el resultado del reentreno tomando una imagen al azar del bache de imágenes de prueba. En el **Anexo E**, se muestran los resultados del reentreno de las redes YOLOv3 y YOLOv3-tiny.

4.1.3 Selección de la plataforma para el reentreno

Antes de continuar es necesario mencionar que entre mayor sea la capacidad computacional de la máquina donde se realizará el entrenamiento más rápido será el reentreno, razón por la cual se decidió aprovechar la herramienta Google Colaboratory, que proporciona alrededor de 360Gbs de espacio de almacenamiento y 12.7Gbs de memoria dedicada con la GPU Nvidia Tesla K80. Una ventaja adicional de este entorno virtual es el manejo de archivos; teniendo en cuenta lo pesado que pueden ser los *dataset*, resulta conveniente no tener que bajarlos al computador y adquirirlos prácticamente al instante.

Se crearon 2 scripts, uno para cada interfaz, en los cuales se usa el almacenamiento de Google Drive para mover archivos, guardar configuraciones, tener acceso rápido a el *dataset* de KITTI, manejar las dependencias y el procesamiento de los datos de cada interfaz.

4.1.3.1 Tensorflow

En la Tabla III se presentan los tiempos de reentreno de las 3 redes de Tensorflow en un computador personal vs Google Colaboratory. Los entrenos se hicieron a 10000 steps:

Tabla III.

Tiempos del reentreno de redes de Tensorflow en un computador portátil vs en Google Colaboratory.

Red	Entreno en pc (Nvidia GeForce 920M)		Entreno en la nube (Nvidia Tesla K80)	
	Tiempo (100 steps) (s)	Tiempo total (s)	Tiempo (100 steps) (s)	Tiempo total (s)
SSD MobileNet V2	135	13500	60	6000
Faster RCNN Inception V2	40	4000	10	1000
Faster RCNN ResNet	78	7800	31	3100

Podemos observar cómo es más conveniente realizar el entreno en la nube, no solo por la velocidad sino también por la conveniencia para la manipulación de datos de bastante peso.

Es necesario mencionar que el reentreno se completó en Google Colaboratory, y localmente solo se corrió hasta que se completaron los primeros 100 *steps*, y se estimó el tiempo total a partir del número total de *steps*.

Una vez realizado los entrenos se procede a probar la velocidad de las redes reentrenadas. En el **Anexo A** se compara la salida de las redes originales con las redes reentrenadas; modificando un poco el script proporcionado por Tensorflow Models podemos probar las redes con una cámara de celular, para ver la velocidad de detección y la fluidez del video, más específicamente los FPS (*frames per second*). En la Tabla IV se presentan los FPS con los que las redes trabajaban de manera fluida y el video no sufre retrasos.

Tabla IV.

FPS con los que las redes de Tensorflow trabajaron de forma fluida.

Red	FPS
SSD MobileNet V2	5.1
Faster RCNN Inception V2	1.7
Faster RCNN ResNet	0.6

De las tres redes reentrenadas la de mayor velocidad es la MobileNet, lo cual no sorprende ya que es una red diseñada para dispositivos móviles. Teniendo en cuenta que solo se probó la velocidad de procesamiento de la detección y no el seguimiento y manejo de esa información, se decidió descartar la “Inception V2” y “ResNet”, teniendo aun como opción la red “MobileNet”.

4.1.3.2 Darknet

Para poder usar la interfaz de Darknet es necesario instalar unas librerías que no están por defecto en Colaboratory, y si se quiere usar la GPU es necesario instalar CUDA, para que Darknet pueda reconocer la tarjeta; luego se comprueba su

funcionamiento realizando una detección, para lo cual se requiere bajar los pesos de las redes (los pesos finales del entrenamiento original), y se usa un comando que Darknet provee.

A continuación, en la Tabla V se presenta la velocidad de entrenamiento de ambas redes hasta los 5000 steps.

Tabla V.

Tiempos del reentreno (de las redes YOLO) en un computador portátil vs en Google Colaboratory.

Red	Entreno en pc (Nvidia GeForce 920M)		Entreno en la nube (Nvidia Tesla K80)	
	Tiempo (1 step) (s)	Tiempo total (s)	Tiempo (1 step) (s)	Tiempo total (s)
YOLO V3	7.4-16	22200-48000	1.7-8	7300
YOLO V3 Tiny	4.3-6	21500-30000	1.7-1.9	10000

Igual que con las redes de Tensorflow, una vez se terminó el reentreno se procedió a comprobar la velocidad de las redes en tiempo real. En la Tabla VI se condensan los resultados de dichas pruebas.

También hay que mencionar que el reentreno local solo se realizó para los primeros 100 steps, y el tiempo total se calculó teniendo en cuenta el step al que llegó el reentreno en la nube; además el tiempo que le tomó a la plataforma para realizar un step varió, por lo que se presenta un rango del tiempo total para el ordenador, mientras que en el de la nube se presenta el tiempo real. Finalmente, Google Colaboratory tiene un tiempo limitado en el que le permite correr al usuario un código (la sesión puede estar abierta de forma continua y se puede escribir y probar código ilimitadamente siempre y cuando ese código no requiera correrse continuamente por un largo periodo de tiempo), pero gracias a que la interfaz de Darknet permite retomar el reentreno (cada que se alcanzan 1000 steps se guarda un previo de la red reentrenada, el cual puede usarse como red final o como punto para seguir el reentreno) se pudo terminar de entrenar la red en 2 sesiones.

Tabla VI.

FPS con los que las redes de Darknet trabajaron de forma fluida.

Red	FPS
YOLOv3	6
YOLOv3-tiny	30

Finalmente se evalúan las redes para comprobar la eficacia de estas. Estos resultados se presentan en la Tabla VII. Las pruebas realizadas en una computadora personal con GPU Nvidia GeForce 920M, de 2 GB de memoria dedicada.

Tabla VII.

Comparación de los FPS y la precisión promedio de las redes de Tensorflow y Darknet.

Red	Fps	IOU (Promedio)	Detecciones (%)	Excelente (%)	Buena (%)	Mala (%)
YOLOv3	5.8	0.785	77.67	10	88.75	1.25
YOLOv3-tiny	29	0.628	84.47	3.45	70.11	26.44
SSD MobileNet V2	5.1	0.698	60.19	6.45	85.48	8.06
Faster RCNN Inception V2	1.7	0.827	87.38	15.56	83.33	1.11
Faster RCNN ResNet	0.6	0.837	80.58	22.89	71.08	6.02

La evaluación se realizó con 20 imágenes, con un total de 103 detecciones, las cuales se etiquetaron manualmente. Al final de la evaluación de cada imagen se sumaron todos los puntajes que arrojó el algoritmo y se hizo un promedio en el que los puntajes IOU van de 0 a 1, siendo 0.5 el umbral para una buena relación entre la detección y su “verdad”. Luego se calcula el porcentaje de buenas detecciones

que tuvo la red (por ejemplo, la red YOLOv3 que obtuvo 77.67%, detectó 80 objetos de entre los 103 que había en las 20 imágenes). Para finalizar en la evaluación se muestra la calidad de las detecciones, distribuidas en 3 categorías (excelente, buena y mala), para lo cual se sumó la cantidad de detecciones con un puntaje mayor o igual a 0.9 (en el caso de excelente), con un puntaje menor a 0.5 (en el caso de mala) y las detecciones con puntajes entre esos dos valores (para el caso de detecciones buenas) y se dividió individualmente con el número total de detecciones que tuvo la red. Por ejemplo, la red YOLOv3 tuvo un total de 80 detecciones, de ellas el 88.75% fueron consideradas buenas, es decir que obtuvimos 71 detecciones con un puntaje por encima de 0.5. En el **Anexo G** se muestran los resultados visuales de la evaluación.

4.2 SUBSISTEMA DE SEGUIMIENTO

En la Fig. 29 se muestra cómo se compone el subsistema de seguimiento.

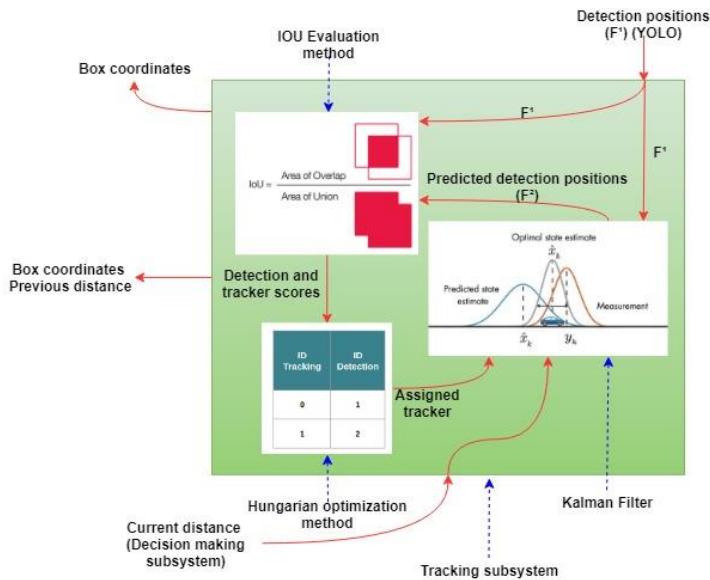


Fig. 29. Subsistema de seguimiento.

4.2.1 Procesamiento en tiempo real

Luego de optar por las redes de YOLO, el siguiente paso es encontrar una manera de usar el detector fuera de la terminal, que nos permita hacer algo con las detecciones, para lo cual se usó el código de prevención de accidentes de tráfico [44]. En este trabajo, ellos usan una red de YOLO para las detecciones utilizando el

filtro de Kalman para realizar seguimiento y usan el método de optimización húngaro junto al IOU con el fin de asignar un filtro a cada detección. Primero se usa un script proporcionado por el repositorio de Darknet, para obtener la localización de las detecciones; en el script “detector.py” se puede ajustar la red que se quiera usar (en nuestro caso Tiny YOLO V3), además se pueden ingresar las clases que se quieren detectar (en nuestro caso carros, buses y camiones) y el threshold (la certeza que tiene el detector de que una detección es correcta). Los *frames* son capturados y enviados al detector por medio de OpenCV, más específicamente por el comando VideoCapture, el cual mientras existan *frames* estará constantemente leyéndolos y estos serán enviados al código principal para su procesamiento.

Finalmente se calculan los FPS a los que se está trabajando, para hacer los cambios dinámicos y no tener que acceder al código cada que se quiera cambiar, y el video final se graba por medio del comando VideoWriter de OpenCV, realizando una codificación MP4.

4.2.2 Implementación del seguidor

Para obtener un seguidor de múltiples objetos es necesario no solo un algoritmo que pueda predecir dónde estarán las detecciones (filtro de Kalman), sino también sortear las detecciones y asignarlas a los filtros correctos (Algoritmo húngaro + IOU). En el **Anexo D** se puede observar un ejemplo del filtro de Kalman.

Cada que se encuentra una nueva detección se crea una nueva instancia de filtro de Kalman, entonces, sí en el primer *frame* se obtuvo una detección se crea la primera instancia, luego sí en la segunda hay 2 detecciones se tiene que ver si alguna de ellas puede ser del mismo objeto detectado en el primer *frame*. Aquí es donde entra el método húngaro de optimización, el cual por medio de puntajes determina si una detección corresponde a la misma en un instante anterior o si es una detección completamente nueva.

Para obtener estos puntajes se recurre a un método de evaluación IOU (Intersection Over Union), el cual compara las cajas de detecciones y, dependiendo de si se entrelazan, determina si las detecciones tienen relación o no. En la Fig. 30 se da un ejemplo del método IOU.

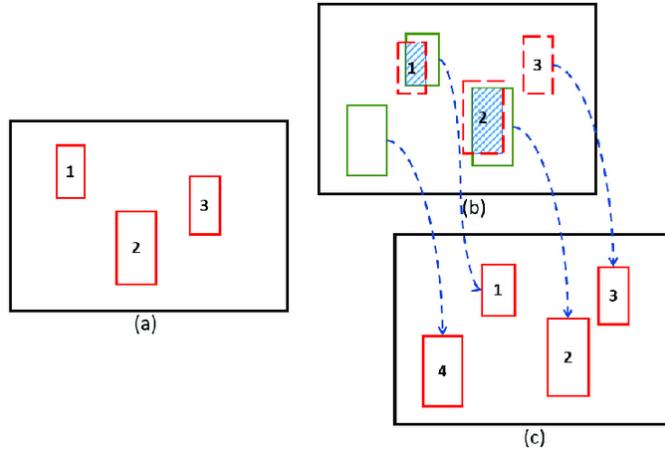


Fig. 30. Ejemplo del algoritmo IOU [45].

Podemos observar a la izquierda el primer *frame*, donde se encontraron 3 detecciones, a la derecha superior el segundo *frame*, en donde los cuadros punteados son las detecciones del *frame* anterior, y los cuadros con líneas completas son las detecciones del *frame* actual. Finalmente, en la parte inferior derecha tenemos el resultado del algoritmo, en el que los cuadros 1 y 2 que aparecieron en ambos *frames* conservaron su identidad, ya que el algoritmo determinó que su solapamiento era suficiente para estar seguro de que era la misma detección.

Luego de obtener estos puntajes se procedió a sortearlos con el algoritmo húngaro, el cual calcula el costo mínimo de asociación de los filtros con las detecciones, de forma que estas seguirán con el mismo filtro mientras existan; las detecciones nuevas no quedarán asignadas, lo que forzará a la creación de una nueva instancia de filtro.

También es necesario estimar un periodo de tiempo para que los filtros desaparezcan si no tienen nuevas detecciones; es decir, si una detección con instancia de *filtro N* deja de aparecer en la escena, este filtro quedará sin detección, y si bien puede predecir dónde estará esta detección, mientras más tiempo pase sin esta más imprecisa será la predicción. Pero que la detección desaparezca no quiere decir que se haya ido por completo, ya que puede que otra detección esté ocultándola, por lo que se espera algunos *frames* para ver si dicha detección vuelve a aparecer, y de no hacerlo la instancia del filtro será liberada de la memoria.

Por último, en la Tabla VIII. se presenta la evaluación del seguidor respecto a su precisión y exactitud, usando la librería pymot [46]. La precisión se mide con la métrica Multiple Object Tracking Precisión (MOTP), mientras que la exactitud es medida con la métrica Multiple Object Tracking Accuracy (MOTA). MOTP es el error de posición total entre los pares de objetos y detecciones de todas las frames, promediado por el número total de coincidencias que el seguidor haya tenido. Muestra la habilidad del seguidor de estimar posiciones precisas de objetos, independientemente de su habilidad para reconocer los objetos y mantener trayectorias consistentes; MOTA contabiliza todos los errores de configuración hechos por el seguidor, los falsos positivos, las pérdidas de objetos y las malas asignaciones. Es decir que el MOTA mide el desempeño del seguidor para mantener trayectorias precisas [47].

Tabla VIII.

Evaluación del seguidor por medio de las metricas MOTP y MOTA.

Results	
Ground truths	1431
False positives	117
Misses	0
Mismatches	0
Recoverable mismatches	0
Non recoverable mismatches	0
Correspondences	1314
MOTP	0,94
MOTA	0,92

Como se observa en la Tabla VIII. el seguidor tuvo un rendimiento bastante bueno, al obtener un puntaje de 94% MOTP y de 92% MOTA. Cabe mencionar que el seguidor se evaluó sobre un video que dura 51 segundos y que se grabó a 25 FPS [48], el número total de frames en el video es por ende de 1275, con un promedio de 1.12 asignaciones por frame.

4.3 SUBSISTEMA DE TOMA DE DECISIONES

En la Fig. 31 se muestra cómo se compone el subsistema de decisiones.

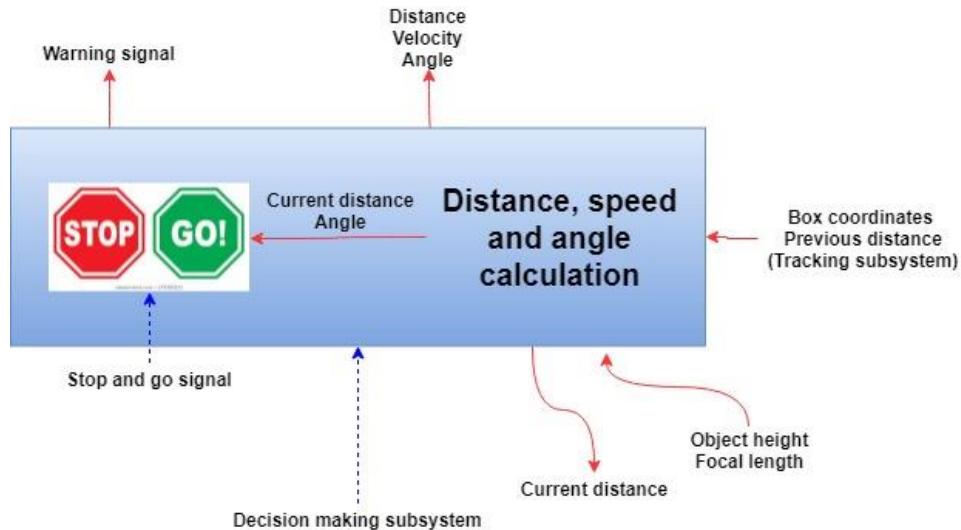


Fig. 31. Subsistema de decisiones.

Luego de obtener la detección y hacer el seguimiento respectivo se implementó un algoritmo que permitiera determinar la distancia que hay desde el ego-car hasta los carros que se están vigilando, ya que en un carro autónomo esto es muy importante para saber cuándo tomar acciones.

Primero se buscó calcular las distancias implementando un cambio de perspectiva, para obtener una imagen en la que la carretera no sea un trapecio sino un rectángulo, pues cuando se observa la carretera desde nuestra perspectiva parece que mientras más lejos se enfoque más angosta es, lo cual no es cierto, y si se viese desde arriba se observaría del mismo tamaño de principio a fin.

Para realizar este cambio de perspectiva en tiempo real primero se tiene que calcular la matriz de transformación, y dicha matriz se calcula tomando 4 puntos (dichos puntos en una tarjeta serían las 4 esquinas, en una carretera serían 4 puntos que siendo vistos desde lo alto deberían ser rectos, pero que en la imagen no lo parecen).

A continuación en la Fig. 32, 33 y 34 se presentan 2 ejemplos: uno donde el cambio de perspectiva es bastante notable, allí se aprecia en la imagen original una carta con un signo de alto, que además de estar girada la foto fue tomada mirando hacia el frente de la mesa, por lo que la parte inferior de la carta parece ser más amplia que la parte superior; el segundo ejemplo es la imagen de una carretera, donde se puede apreciar a lo que se refiere con trapecio a la hora de señalar los puntos de trabajo y se puede observar que la carretera parece disminuir su tamaño a medida que avanza. (la imagen original de la Fig. 33 fue tomada de un video del programa *Become a Self-Driving Car Engineer*, de Udacity, y fue procesada manualmente, la Fig. 34 fue el resultado de dicho procesamiento):

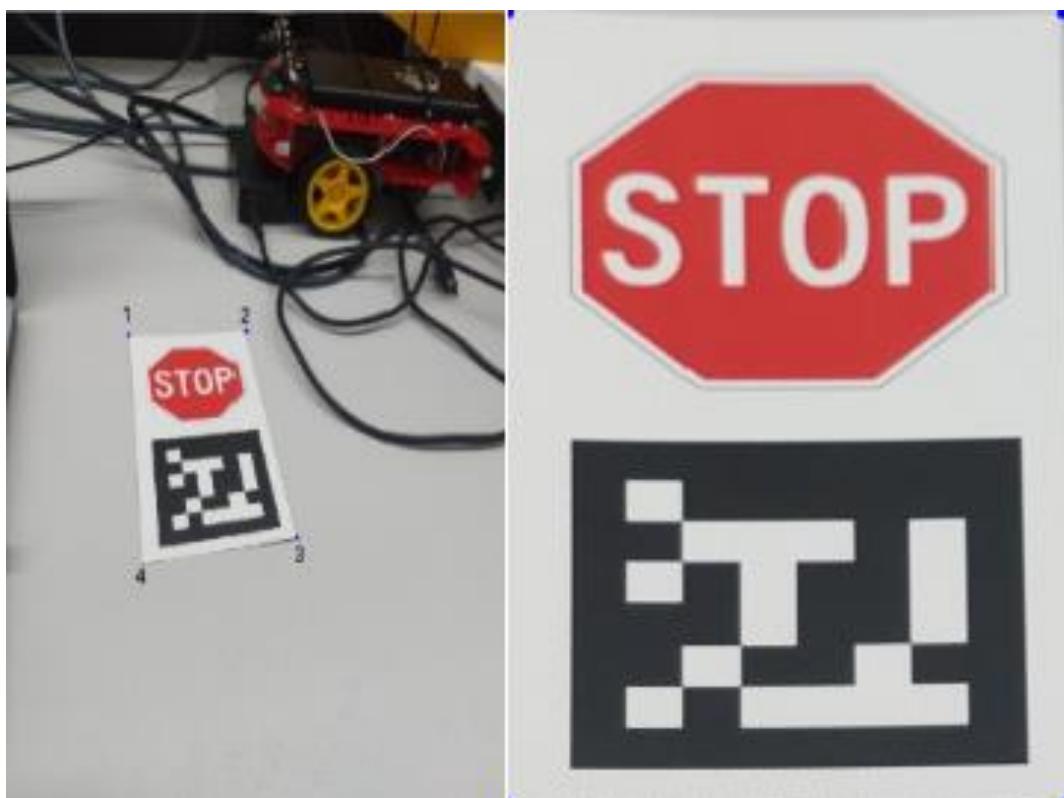


Fig. 32. Toma de puntos y resultado final.



Fig. 33. Ejemplo de selección de puntos [48].

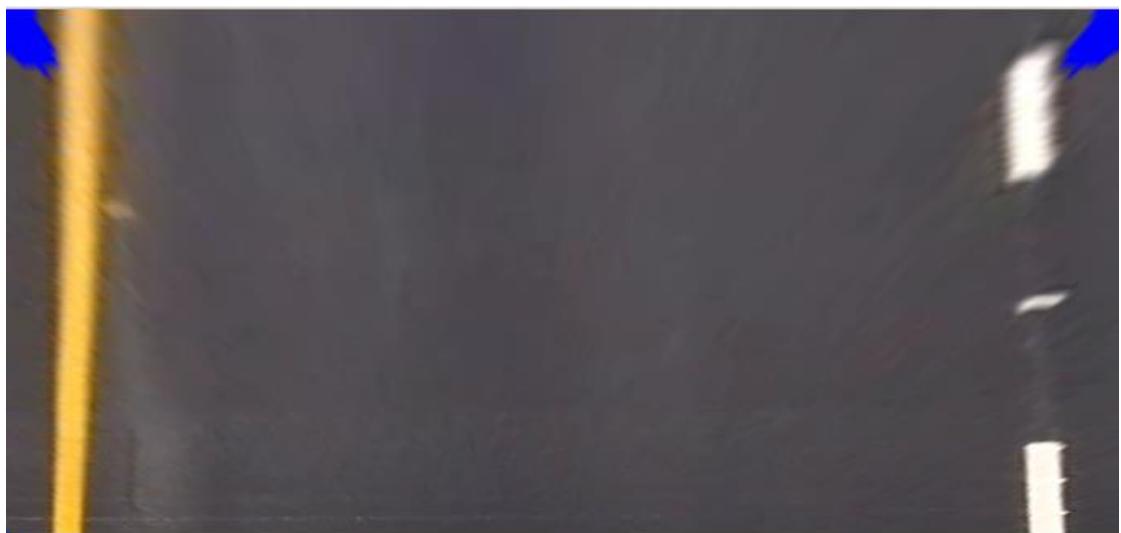


Fig. 34. Resultado final del cambio de perspectiva.

Si bien en el primer ejemplo se puede apreciar el cambio de perspectiva de forma clara, en el segundo la carretera parece más corta de lo que es, esto debido a la relación entre el tamaño de la imagen y la longitud del trayecto que se tomó, ya que entre más nos alejamos para tomar la parte superior del trapecio, más larga será la carretera, pero dicho aumento no es proporcional a los píxeles de imagen que representarán esa parte del trayecto.

Algo importante para tener en cuenta es que los puntos deben tomarse empezando por la parte superior izquierda, y siguen en sentido de las manecillas del reloj, como se especificó en el código.

Luego de obtener la matriz de transformación, la idea era transformar el punto inferior de la detección de los carros con respecto al centro de esa parte de la imagen para obtener la distancia de la detección respecto al ego-car. Pero dichos resultados fueron infructuosos ya que los valores no eran razonables.

La segunda técnica empleada fue la del triángulo de similitud, para el cual es necesario conocer la longitud o el ancho del objeto al que se le quiere calcular la distancia, y requiere de calibrar la cámara para hallar la longitud focal de esta, la cual nos permitirá por regla de 3 obtener la distancia. Para esto se asumió la altura de los carros en 1.5 metros y de los camiones en 2.8 metros. Se trabajó con la altura ya que esta siempre variará proporcionalmente en las detecciones, mientras que el ancho dependerá del ángulo en que se vea el carro, así como también de si se ve completo o parcialmente.

Finalmente se calcula la altura en píxeles de las detecciones, tomando los puntos inferior y superior de estas, y se calcula la distancia a partir de la fórmula de la longitud focal (6), donde F es la longitud focal, P es la altura o el ancho aparente, D es la distancia que se busca y W es el ancho o la altura real.

$$F = (P * D) * W \quad (6)$$

De acuerdo con lo anterior es necesario conocer D (7) para hallar la longitud focal. Primero se calibra la cámara tomando una foto donde se encuentre el objeto al que se le quiere calcular la distancia, luego se mide la distancia real en esa foto y se calcula la longitud focal, al final se despeja la fórmula de forma que la distancia sea la variable por encontrar.

$$D = \frac{F}{P*W} \quad (7)$$

En el **Anexo B** se presenta la evaluación del método con imágenes propias.

Luego de hallar la distancia se calcula el ángulo de la detección, con el cual sabremos en qué dirección del ego-car está ubicada, así como también para saber si esta detección está muy cercana y si se deberán entonces tomar acciones.

Para hallar el ángulo se calcula la tangente inversa entre los catetos del triángulo, que se forma trazando una línea desde la parte inferior central del fotograma y la parte inferior central del cuadro de la detección.

Aprovechando la estructura del seguidor se calcula la velocidad relativa a la que va el carro, guardando la distancia de la detección en el *frame* n, y luego sacando la diferencia entre la distancia en el *frame* n+1 y la distancia en el *frame* n. Finalmente se divide el resultado por 1/FPS que es la velocidad con la que se toman los *frames*; esta velocidad será con respecto a la que lleva el ego-car: si la velocidad es 0 quiere decir que ambos carros llevan la misma velocidad, mientras que si es negativa el vehículo se está moviendo más rápido que la detección.

Para terminar, si una detección está a una distancia menor o igual a 3 metros y con un ángulo entre 75 y 105°, se crea una regla, la cual mostrará una advertencia en la pantalla.

A continuación, en la Fig. 35, 36 y 37 se presentan los resultados de la implementación del algoritmo (las imágenes originales fueron tomadas de un video del programa *Become a Self-Driving Car Engineer*, de Udacity, y fueron procesadas manualmente):



Fig. 35. Resultado del algoritmo de medición de distancias y ángulos ($t=0$) [48].



Fig. 36. Resultado del algoritmo de medición de distancias y ángulos ($t=1$) [48].



Fig. 37. Resultado del algoritmo de medición de distancias y ángulos (t=2) [48].

Observamos que en la Fig. 35 existe una detección a una distancia mayor a 3 metros, por lo que dentro de nuestros parámetros es seguro avanzar; en la Fig. 36 se muestra como entra en escena una segunda detección a una distancia menor a 3 metros, por lo que el algoritmo indica que lo mejor es disminuir la velocidad para alejarnos de ella; por último, en la Fig. 37 se ve que ambas detecciones están a una distancia prudente, lo que nos vuelve a poner el sistema como seguro para avanzar.

Para propósitos de evaluación se sobrepasó el límite de angulación de la regla creada, con el fin de observar el cambio de la advertencia tipo semáforo que se implementó.

4.4 ACOPLAMIENTO DEL SISTEMA A UNA TARJETA DE DESARROLLO JETSON TK1

Para probar el sistema en tiempo real se tiene una tarjeta de desarrollo Nvidia Jetson TK1, la cual actuará como computador y correrá los algoritmos dentro del carro. Antes de comenzar, se formatea la tarjeta para borrar los datos que no son necesarios, abriendo espacio para las librerías que se utilizarán. Se puede reinstalar el sistema operativo de 2 formas: por medio de comandos en la terminal (este método solo permite adquirir el sistema operativo Ubuntu 14.04) e instalando JetPack, el cual es un paquete disponible para tarjetas Jetson que no solo provee

el sistema operativo, sino también herramientas como CUDA y opencv4tegra, la librería de OpenCV lista para usar en las tarjetas Jetson.

El primer método es más rápido; sin embargo, el segundo es más eficaz, ya que evita que el usuario tenga que buscar e instalar por sí mismo estas herramientas.

La tarjeta Jetson TK1 está limitada a usar la versión 6.5 de CUDA, plataforma que permite usar en toda su capacidad las tarjetas de video dedicadas de Nvidia. En el proceso esto presentó un problema, porque al compilar el repositorio de Darknet original se generaron errores al no estar definidos los métodos implementados en las librerías faltantes, ya que la versión de CUDA requerida para usar la interfaz de Darknet es mayor o igual a 9, debido a que esta usa librerías implementadas en versiones modernas de CUDA que simplifica operaciones matemáticas.

Para solucionar este problema se buscó un repositorio que pudiese construirse con versiones antiguas de CUDA, encontrándose uno [49] que se basó en el repositorio original de Darknet con el propósito de implementarlo para arquitecturas con CUDA 6.5.

Antes de construir el repositorio es necesario instalar OpenCV (una librería open source de visión artificial), pues con esta se manipularán las imágenes. Se optó por instalar manualmente OpenCV debido a que la versión que viene en JetPack es la 2.4 y para no tener que modificar todo el código la versión mínima requerida es 3.4.

Para instalar OpenCV en su versión 3.4 se requieren de al menos 8 GBs de memoria libre, los cuales la tarjeta no disponía, por lo que existen 3 opciones para proceder: usar una memoria USB; sino, una microSD como almacenamiento temporal para los archivos de construcción; o en su caso emplear el computador como servidor para guardar temporalmente los archivos de construcción. Debido a la falta de memorias se optó por usar el ordenador como un servidor, por medio de NFS (network file system), sistema que permite montar una carpeta del computador servidor a un sistema usuario, que dándole permisos de super usuario puede modificar la carpeta como si esta estuviera dentro del mismo sistema. Esto también facilitó la transferencia de archivos del ordenador a la tarjeta Jetson.

Una vez solucionados los problemas de dependencias se construye el repositorio de Darknet para TK1 realizando unos cambios en el archivo Makefile, para que este trabaje con la GPU y con soporte para OpenCV.

Luego de construir el repositorio se comprueba su funcionamiento, para lo cual se descargan los pesos de las redes Yolo V3 y Tiny Yolo V3 de la página de soporte de Darknet [43]. Finalmente se corre el código del proyecto en la tarjeta, que nos indica que hace falta instalar los codificadores para grabar video, los cuales no son soportados en la versión de sistema operativo Ubuntu 14.04 debido a que su soporte oficial termina, por lo que fue necesario actualizar el sistema a Ubuntu 16.04.

Al ser necesario una cámara y no contar con una cámara web, se optó por usar la cámara del celular de forma inalámbrica, por medio de la aplicación **Ip Webcam** [40], con la cual el celular se vuelve un servidor que puede ser accedido desde una dirección ip, y que nos permite entre otras cosas ajustar la resolución y los FPS del video. Debido a que el carro no cuenta con internet es necesario usar un segundo celular que actúe como router, puesto que, para que la aplicación funcione es necesario que ambos celular-cámara y la tarjeta Jetson TK1 se encuentren conectados a la misma red.

A continuación, en la Fig. 38 y 39 se muestra un ejemplo de la aplicación Ip Webcam. En la parte inferior se encuentra la Ip a la que nos conectaremos.



Fig. 38. Uso de la aplicación Ip Webcam.

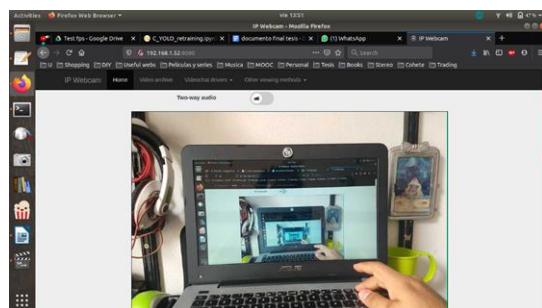


Fig. 39. Conexión a la plataforma web de Ip Webcam.

Debido a que la tarjeta requiere de una fuente de poder de 12V constantes y al menos 1 amperio, se utilizó unas de las salidas instaladas en los automóviles convencionales para extraer esa cantidad de voltaje a través un adaptador, que posee una entrada removible que se utiliza según el puerto que se requiera para la operación.

En la Fig. 40 se muestra el adaptador para el carro, en la Fig. 41 se muestra la salida del carro, en la Fig. 42 se muestra la salida del adaptador (adaptable) y en la Fig. 43 se muestra la tarjeta en funcionamiento.



Fig. 40. Adaptador para el carro.



Fig. 41. Salida del carro.

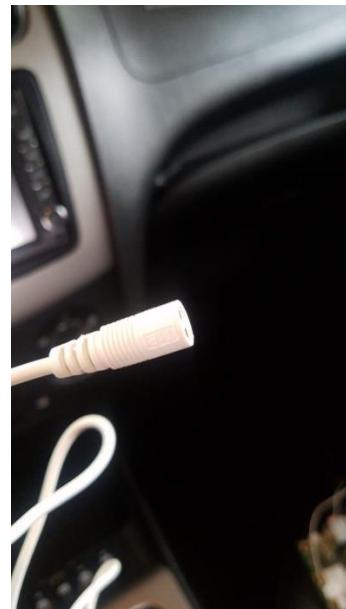


Fig. 42. Salida del adaptador.

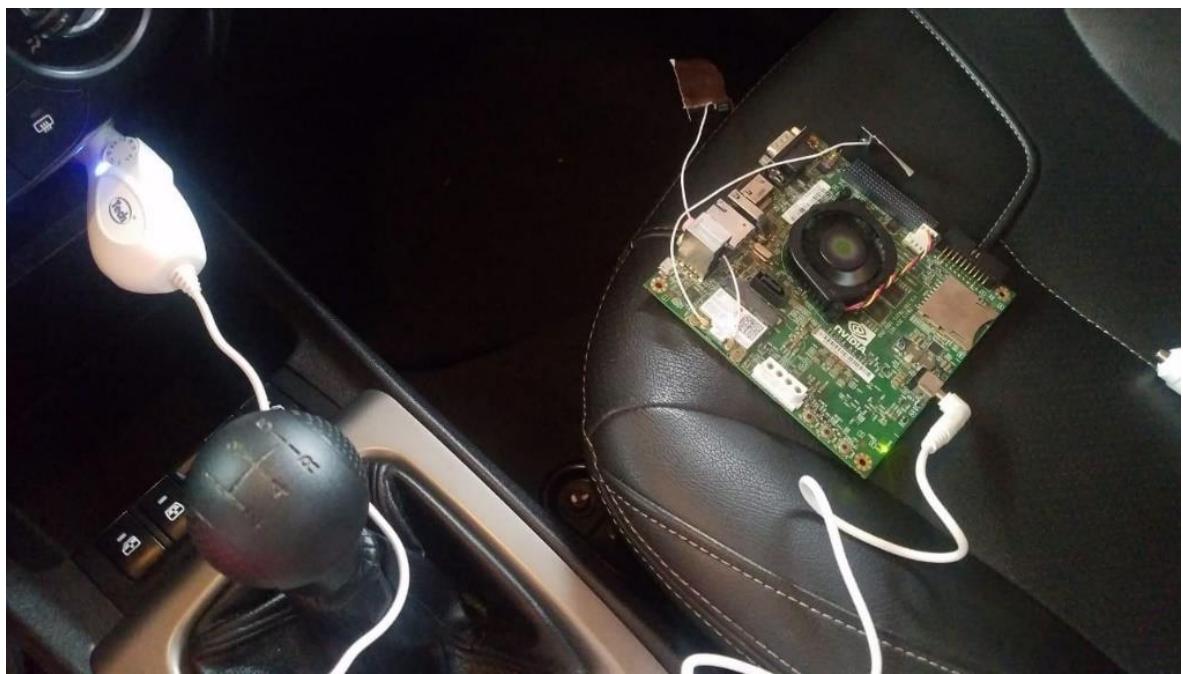


Fig. 43. Tarjeta embebida en funcionamiento.

Además de esto y teniendo en cuenta que hasta ahora se ha trabajado la tarjeta Jetson TK1 con un display, es necesario manejarla de forma remota con un

computador, accediendo a ella por medio de el comando ssh (un protocolo de administración remota); además, por medio del protocolo de transferencia de archivos de ssh (sftp) podemos ver y modificar los archivos de la tarjeta desde el computador para poder observar los videos grabados durante la prueba.

Finalmente se realiza una primera prueba con la cámara del celular, manejando la tarjeta desde el computador para ajustar la configuración del video. Teniendo en cuenta que se quiere obtener un resultado en tiempo real, es necesario que la tarjeta trabaje acorde a las especificaciones dadas previamente, por lo que cuando se ejecuta el código se espera que lo que se ve en la cámara se muestre en la salida de la pantalla, teniendo en cuenta esto se ajustan los FPS para obtener dicho resultado, con lo que nos dimos cuenta de que en un principio solo podíamos considerar una salida en tiempo real con 2 FPS, un número muy bajo (esto indica que en 1 segundo se enviaran 2 *frames* de video, por lo que se percibirá el video como cortado).

Subir los FPS no era una opción de momento ya que el video no solo quedaba más corto de lo que se había grabado, sino que también en ocasiones “saltaba”, por lo que se consideraron otras opciones. Una de esas opciones fue bajar la resolución del video y aumentar un poco los FPS, el resultado fue mejor, pero la calidad del video grabado bajo considerablemente. Finalmente, se decidió por bajar el tamaño de la entrada de la red (que en un principio estaba en 416x416) a 288x288 pixeles. Esto nos permitió alcanzar hasta 5 FPS con poco recorte de video.

Teniendo en cuenta que se quiere probar la red en tiempo real, se usa un *batch* y subdivisión de 1, para aligerar la carga computacional. Hay que aclarar que el problema con cambiar la entrada de la red por una más pequeña es que precisión de la red para detectar disminuye, por lo que se tuvo que encontrar un balance entre el tamaño de entrada y los FPS.

A continuación, en la Fig. 44, 45, 46, 47 y 48 se presenta una pequeña porción del video resultado, en donde las imágenes son *frames* tomadas una seguida de otra, para validar el funcionamiento del seguidor.



Fig. 44. Seguimiento y medición de la distancia de un carro (t=0).



Fig. 45. Seguimiento y medición de la distancia de un carro (t=1).



Fig. 46. Seguimiento y medición de la distancia de un carro (t=2).

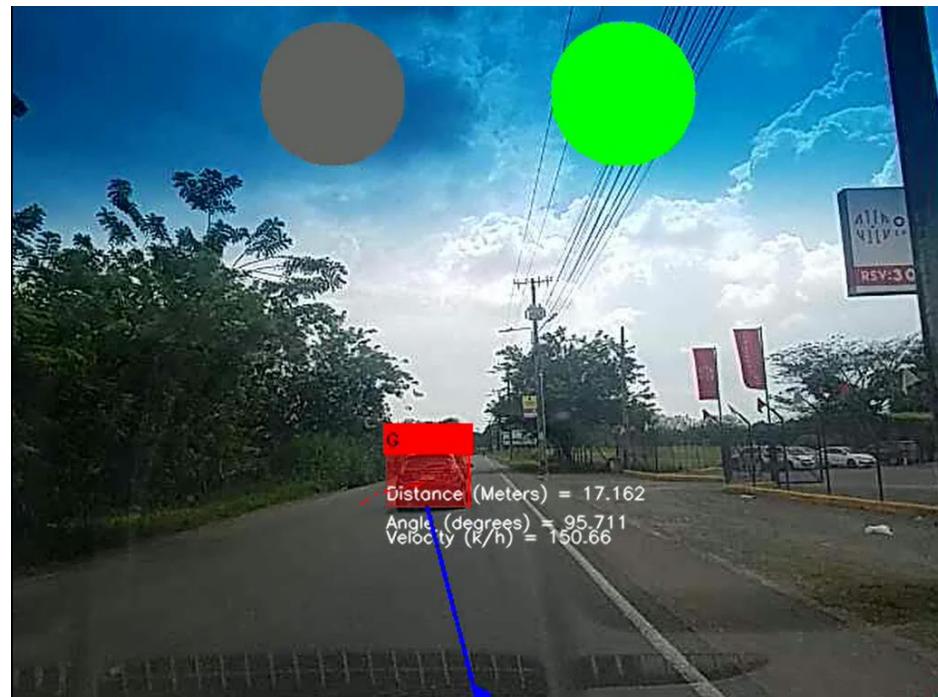


Fig. 47. Seguimiento y medición de la distancia de un carro (t=3).



Fig. 48. Seguimiento y medición de la distancia de un carro (t=4).

En el **Anexo C**, se presentan más resultados de la prueba de campo.

5. CONCLUSIONES

El desarrollo de este trabajo permitió conseguir de manera exitosa la detección y seguimiento de vehículos en carretera, en tiempo real. Donde el algoritmo es capaz de generar una detección y seguimiento distinguiendo la cantidad de carros que estén transitando la carretera, mostrando diferentes cuadros de distintos colores para poder distinguir los diferentes carros en movimiento. Este trabajo pretende brindar un acercamiento al problema de la autonomía de conducción de vehículos para que en futuros trabajos se pueda instalar en vehículos y lograr una autonomía exitosa.

Al modificar parámetros como la resolución de la imagen, los FPS y el tamaño de entrada de la red se logró una mejor precisión en los resultados que se querían obtener, como por ejemplo en la altura y en el ancho. Al realizar pruebas se tuvieron que modificar estos parámetros mencionados en ocasiones para lograr una mejor efectividad en la detección. Se tuvo que encontrar un balance entre el tamaño de entrada y los FPS puesto que, al disminuir la entrada de la red, las detecciones se tornan menos precisas.

Los resultados arrojados al evaluar el entrenamiento de las redes MobileNet, Faster RCNN Inception V2, Faster RCNN ResNet y YOLO, demostraron que es mejor utilizar YOLO ya que tiene una mayor rapidez y precisión en la detección de automóviles.

El filtro de Kalman es muy útil para este tipo de trabajos, no solo porque logra una precisión bastante buena, sino porque también es computacionalmente ligero, lo que nos ayudó a poder implementar el algoritmo en tiempo real.

Aunque se lograron obtener resultados buenos, la tarjeta Jetson TK1 no es la adecuada para realizar este tipo de trabajos, ya que se podría considerar obsoleta con respecto a nuevos modelos que ya han salido que tienen una relación calidad/precio es buena.

Como trabajo futuro se puede pensar en realizar fusión sensorial, para que no solo se cuente con una cámara para realizar detección, sino también con otros sensores como los radares y láseres teniendo en cuenta que un vehículo autónomo cuenta con una multitud de sensores y que todos están enviando información al tiempo y que esta información debe ser procesada de forma conjunta para poder obtener una visión completa del entorno.

BIBLIOGRAFÍA O REFERENCIAS

- [1] Smith, Matthew. "The number of cars worldwide is set to double by 2040". [En línea]. [Consultado: 20 de enero de 2020]. Disponible en: <https://www.weforum.org/agenda/2016/04/the-number-of-cars-worldwide-is-set-to-double-by-2040>
- [2] B. I. I. Intelligence BI, "10 million self-driving cars will be on the road by 2020", Business Insider. [En línea]. [Consultado: 19 de septiembre de 2018]. Disponible en: <https://www.businessinsider.com/report-10-million-self-driving-cars-will-be-on-the-road-by-2020-2015-5-6>.
- [3] Reyes Fajardo, Juan Manuel. "Vehículos autónomos, ¿a la vuelta de la esquina?". [En línea]. [Consultado: 19 de septiembre de 2018]. Disponible en: <https://www.publimetro.co/co/tacometro/2018/01/18/vehiculos-autonomos-la-vuelta-la-esquina.html>.
- [4] A. Agarwal y S. Suryavanshi, «Real-Time* Multiple Object Tracking (MOT) for Autonomous Navigation», p. 5.
- [5] Girshick, Ross. (2015). Fast r-cnn. 10.1109/ICCV.2015.169.
- [6] Held, David & Thrun, Sebastian & Savarese, Silvio. (2016). Learning to Track at 100 FPS with Deep Regression Networks. LNCS. 9905. 749-765. 10.1007/978-3-319-46448-0_45.
- [7] D.-A. Beaupré, G.-A. Bilodeau, y N. Saunier, «Improving Multiple Object Tracking with Optical Flow and Edge Preprocessing», arXiv:1801.09646 [cs], ene. 2018.
- [8] Yang, Yuebin & Bilodeau, Guillaume-Alexandre. (2017). Multiple Object Tracking with Kernelized Correlation Filters in Urban Mixed Traffic. 209-216. 10.1109/CRV.2017.18.
- [9] J. Jodoin, G. Bilodeau and N. Saunier, «Urban Tracker: Multiple object tracking in urban mixed traffic», IEEE Winter Conference on Applications of Computer Vision, Steamboat Springs, CO, 2014, pp. 885-892.

- [10] L. Ding y A. Goshtasby, «On the Canny edge detector», Pattern Recognition, vol. 34, No. 3, pp. 721-725, mar. 2001.
- [11] A. K. Chauhan y P. Krishan, «Moving Object Tracking using Gaussian Mixture Model and Optical Flow», International Journal of Advanced Research in Computer Science and Software Engineering, p. 4, 2013.
- [12] R. S. Rakibe y B. D. Patil, «Background Subtraction Algorithm Based Human Motion Detection», vol. 3, No. 5, p. 4, 2013.
- [13] Li, Chenge Lexi & Dobler, Gregory & Feng, Xin & Wang, Yao. (2019). TrackNet: Simultaneous Object Detection and Tracking and Its Application in Traffic Video Analysis.
- [14] Hou, Rui & Chen, Chen & Shah, Mubarak. (2017). An End-to-end 3D Convolutional Neural Network for Action Detection and Segmentation in Videos. 14.
- [15] Simonyan, Karen & Zisserman, Andrew. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv 1409.1556.
- [16] Jaderberg, Max & Simonyan, Karen & Zisserman, Andrew & Kavukcuoglu, Koray. (2015). Spatial Transformer Networks. Advances in Neural Information Processing Systems 28 (NIPS 2015).
- [17] R. Hou, C. Chen, y M. Shah, «Tube Convolutional Neural Network (T-CNN) for Action Detection in Videos», en 2017 IEEE International Conference on Computer Vision (ICCV), Venice, 2017, pp. 5823-5832.
- [18] Dong, Z.; Wu, Y.; Pei, M.; Jia, Y. Vehicle type classification using a semisupervised convolutional neural network. IEEE Trans. Intel. Transp. Syst. 2015, 16, 2247–2256.
- [19] Sang, Jun & Wu, Zhongyuan & Guo, Pei & Hu, Haibo & Xiang, Hong & Zhang, Qian & Cai, Bin. An improved YOLOv2 for vehicle detection. Sensors. [Consultado: 16 de enero de 2020]. vol. 18, doi: 10.3390/s18124272.

- [20] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, “The Cityscapes Dataset for Semantic Urban Scene Understanding,” in Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.
- [21] Arróspide, Jon & Salgado, Luis & Nieto, Marcos. Vehicle Image Database. [Consultado: 16 de enero de 2020]. Disponible en: https://www.gti.ssr.upm.es/data/Vehicle_database.html
- [22] Geiger, Andreas & Lenz, Philip & Urtasun, Raquel. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. 2012. [Consultado: 16 de enero de 2020]. Disponible en: http://www.cvlibs.net/datasets/kitti/eval_object.php
- [23] Sudarshan & Adhikari, Anup & Koirala, Binish & Bhattacharya, Sparsha. Vehicles - Nepal. 2017. [Consultado: 16 de enero de 2020]. Disponible en: <https://www.kaggle.com/sdevkota007/vehicles-nepal/metadata>.
- [24] Krause, Jonathan & Stark, Michael & Deng, Jia & Fei-Fei, Li. Cars Dataset. [Consultado: 16 de enero de 2020]. Disponible en: https://ai.stanford.edu/~jkrause/cars/car_dataset.html
- [25] Lin, Tsung-Yi & Maire, Michael & Belongie, Serge & Bourdev, Lubomir & Girshick, Ross & Hays, James & Perona, Pietro & Ramanan, Deva & Zitnick, Lawrence & Dollár, Piotr. Microsoft COCO: Common Objects in Context. 2015. [Consultado: 16 de enero de 2020]. Disponible en: <https://arxiv.org/abs/1405.0312>.
- [26] Saha, Sumit. “A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way”. [En línea]. [Consultado: 6 de noviembre de 2019]. Disponible en: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>
- [27] Howard, Andrew & Zhu, Menglong & Chen, Bo & Kalenichenko, Dmitry & Wang, Weijun & Weyand, Tobias & Andreetto, Marco & Adam, Hartwig. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.

- [28] Mansouri, Ilias. Computer Vision Part 4: An overview of Image Classification architectures. 2019. [Consultado: 16 de enero de 2020]. Disponible en: <https://medium.com/overture-ai/part-4-image-classification-9a8bc9310891>.
- [29] Sandler, Mark & Howard, Andrew & Zhu, Menglong & Zhmoginov, Andrey & Chen, Liang-Chieh. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. 4510-4520. 10.1109/CVPR.2018.00474.
- [30] Ren, Shaoqing & He, Kaiming & Girshick, Ross & Sun, Jian. (2016). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. 1-10.
- [31] Szegedy, Christian & Liu, Wei & Jia, Yangqing & Sermanet, Pierre & Reed, Scott & Anguelov, Dragomir & Erhan, Dumitru & Vanhoucke, Vincent & Rabinovich, Andrew. (2015). Going deeper with convolutions. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 1-9. 10.1109/CVPR.2015.7298594.
- [32] Szegedy, Christian & Vanhoucke, Vincent & Ioffe, Sergey & Shlens, Jon & Wojna, ZB. (2016). Rethinking the Inception Architecture for Computer Vision. 10.1109/CVPR.2016.308.
- [33] He, Kaiming & Zhang, Xiangyu & Ren, Shaoqing & Sun, Jian. (2016). Deep Residual Learning for Image Recognition. 770-778. 10.1109/CVPR.2016.90.
- [34] Dwivedi, Priya. Understanding and Coding a ResNet in Keras. 2019. [Consultado: 16 de enero de 2020]. Disponible en: <https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33>.
- [35] Redmon, Joseph & Divvala, Santosh & Girshick, Ross & Farhadi, Ali. (2016). You Only Look Once: Unified, Real-Time Object Detection. 779-788. 10.1109/CVPR.2016.91.
- [36] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," p. 16, 2006.

- [37] Zumino, Wesley. "Intuition behind the Hungarian Algorithm". [En línea]. [Consultado: 20 de enero de 2020]. Disponible en: <https://brilliant.org/discussions/thread/intuition-behind-the-hungarian-algorithm/>.
- [38] Rosebrock, Adrian. "Intersection over Union (IoU) for object detection". [En línea]. [Consultado: 20 de enero de 2020]. Disponible en: <https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/>
- [39] Sarkar, Dipanjan. A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning. 2018. [Consultado: 16 de enero de 2020]. Disponible en: <https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a>.
- [40] Pavel Khleovich. (2010). Ip Webcam (1.14.31.737) [Aplicación Móvil]. Disponible en: https://play.google.com/store/apps/details?id=com.pas.webcam&hl=es_419
- [41] David Dao. 2016, Tensorflow models, [GitHub repository]. [Consultado: 10 de mayo de 2019]. Disponible en: <https://github.com/tensorflow/models>
- [42] Redmon, Joseph & Farhadi, Ali. (2018). YOLOv3: An Incremental Improvement.
- [43] Farhadi, Ali. "YOLO: Rea-Time Object Detection". [En línea]. [Consultado: 10 de mayo de 2019]. Disponible en: <https://pjreddie.com/darknet/yolo/>
- [44] Toro, Walter & Perafan, Juan & Mondragon, Oscar & Obando-Ceron, Johan. (2019). Divide and Conquer: an Accurate Machine Learning Algorithm to Process Split Videos on a Parallel Processing Infrastructure.
- [45] Cohen, Jeremy. Computer Vision for tracking. 2019. [Consultado: 16 de enero de 2020]. Disponible en: <https://towardsdatascience.com/computer-vision-for-tracking-8220759eee85>.

- [46] Videmo. 2016, pymot, [GitHub repository]. [Consultado: 18 de marzo de 2020]. Disponible en: <https://github.com/Videmo/pymot>
- [47] Bernardin, Keni & Elbs, Alexander & Stiefelhagen, Rainer. (2006). Multiple object tracking performance metrics and evaluation in a smart Room environment. Proceedings of IEEE International Workshop on Visual Surveillance.
- [48] Udacity. project_video [video]. Udacity. 0:55 minutos. [Consultado: 15 de mayo de 2019]. Disponible en: https://raw.githubusercontent.com/udacity/CarND-Advanced-Lane-Lines/master/project_video.mp4
- [49] dtmoodie. 2016, darknet, [GitHub repository]. [Consultado: 3 de septiembre de 2019]. Disponible en: <https://github.com/dtmoodie/darknet/tree/tk1>
- [50] BML electronics. BML Safe CCTV 8CH - sample view on the street [video]. YouTube. 3:00 minutos. [Consultado: 15 de febrero de 2020]. Disponible en: <https://www.youtube.com/watch?v=FQnhIjvKIZE>
- [51] Bradski, Gary. (2000). The OpenCV library. Dr. Dobb's Journal of Software Tools. 25.
- [52] Rosebrock, Adrian. "Find distance from camera to object/marker using Python and OpenCV". [En línea]. [Consultado: 10 de octubre de 2019]. Disponible en: <https://www.pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/>
- [53] Zuccolo, Ricardo. "Self-driving Cars — Advanced computer vision with OpenCV, finding lane lines". [En línea]. [Consultado: 10 de agosto de 2019]. Disponible en: <https://chatbotslife.com/self-driving-cars-advanced-computer-vision-with-opencv-finding-lane-lines-488a411b2c3d>
- [54] Rosebrock, Adrian. "4 Point OpenCV getPerspective Transform Example". [En línea]. [Consultado: 10 de octubre de 2019]. Disponible en: <https://www.pyimagesearch.com/2014/08/25/4-point-opencv-getperspective-transform-example/>

- [55] AlexeyAB. 2016, darknet, [GitHub repository]. [Consultado: 6 de mayo de 2019]. Disponible en: <https://github.com/AlexeyAB>
- [56] Ssaru. 2018. convert2Yolo, [GitHub repository]. [Consultado: 10 de mayo de 2019]. Disponible en: <https://github.com/ssaru/convert2Yolo>
- [57] Geiger, Andreas & Lenz, P & Stiller, Christoph & Urtasun, Raquel. (2013). Vision meets robotics: the KITTI dataset. *The International Journal of Robotics Research*. 32. 1231-1237. 10.1177/0278364913491297.
- [58] Nowak, Will. "How to Train Your Model (Dramatically Faster)". [En línea]. [Consultado: 10 de mayo de 2019]. Disponible en: <https://towardsdatascience.com/how-to-train-your-model-dramatically-faster-9ad063f0f718>
- [59] Abadi, Martín & Agarwal, Ashish & Barham, Paul & Brevdo, Eugene & Chen, Zhifeng & Citro, Craig & Corrado, G.s & Davis, Andy & Dean, Jeffrey & Devin, Matthieu & Ghemawat, Sanjay & Goodfellow, Ian & Harp, Andrew & Irving, Geoffrey & Isard, Michael & Jia, Yangqing & Jozefowicz, Rafal & Kaiser, Lukasz & Kudlur, Manjunath & Zheng, Xiaoqiang. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems.

ANEXOS

Anexo A. Validación del entrenamiento de las redes de Tensorflow con imágenes del dataset de KITTI

Todas las imágenes originales provienen de [22]:

Imagen original

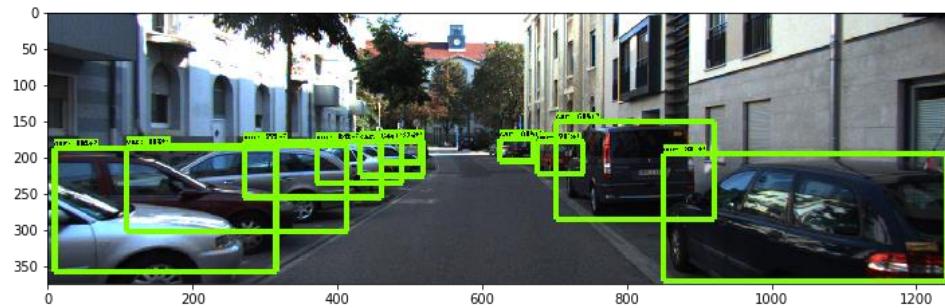


- MobileNet

-Sin reentreno

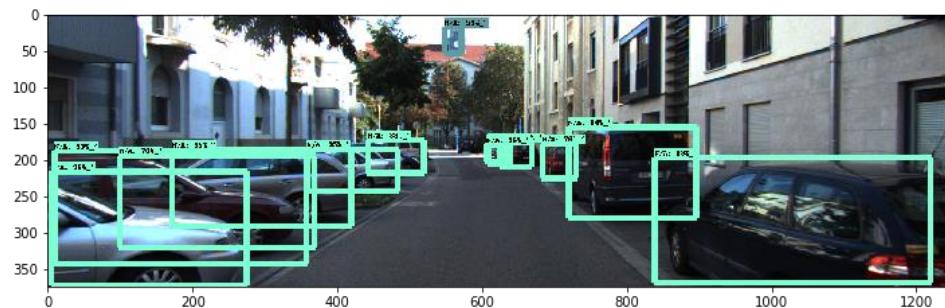


-Reentrenada

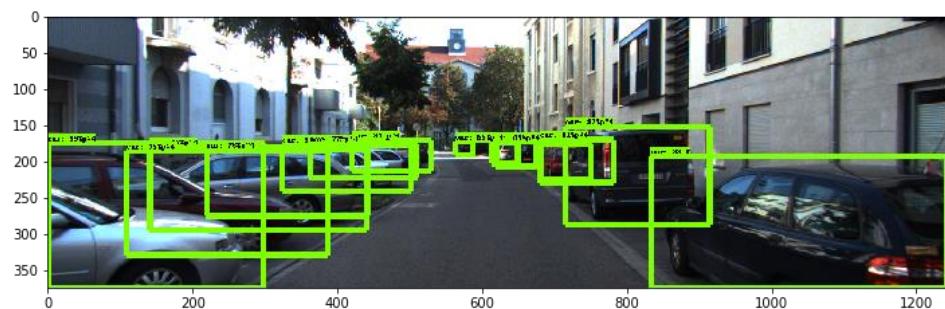


- Inception V2

-Sin reentreno



-Reentrenada



- ResNet

-Sin reentreno



-Reentrenada

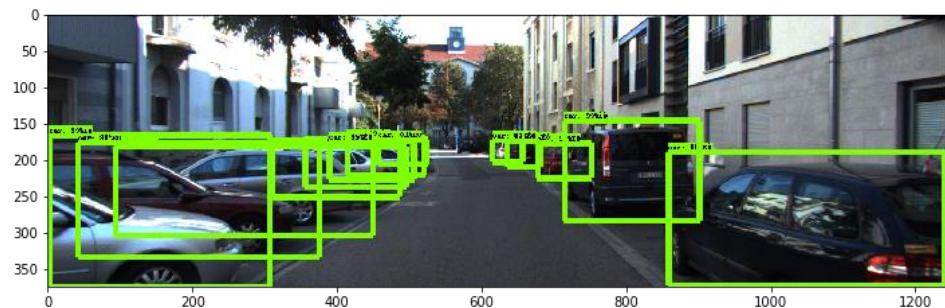
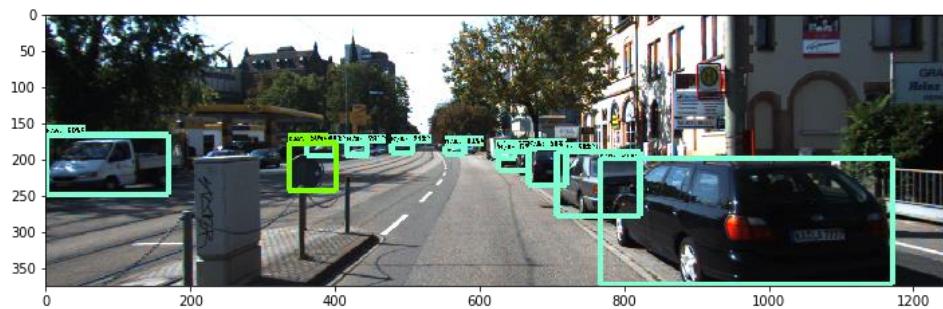


Imagen original

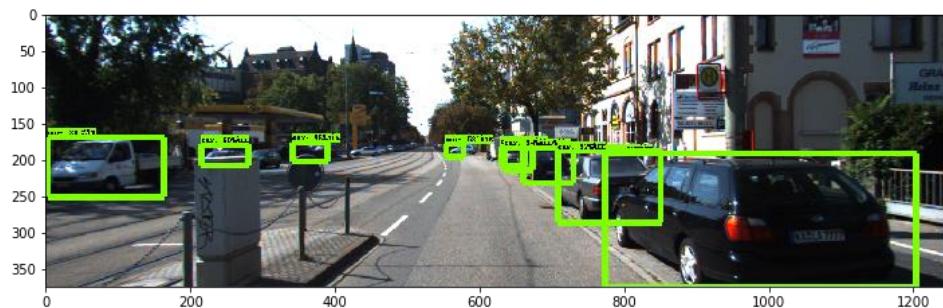


- MobileNet

-Sin reentreno

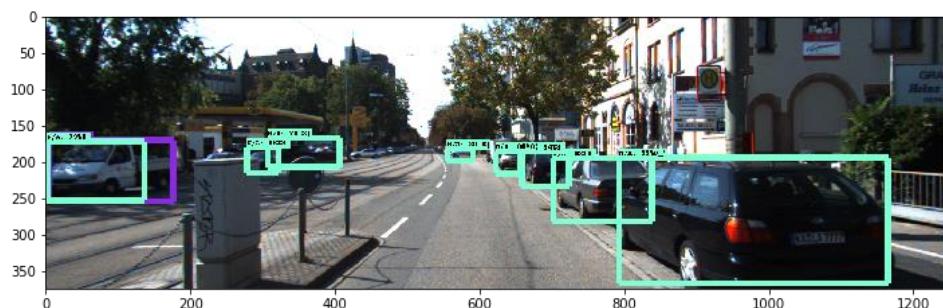


-Reentrenada

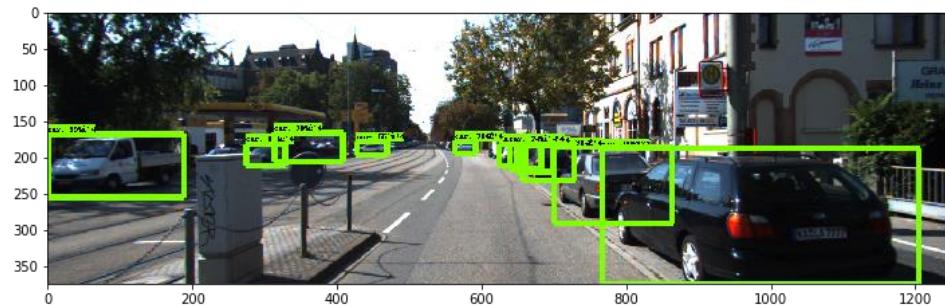


- Inception V2

-Sin reentreno

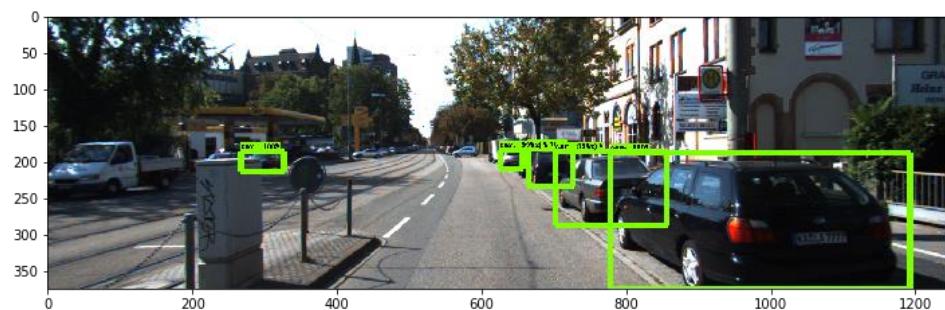


-Reentrenada



- ResNet

-Sin reentreno



-Reentrenada

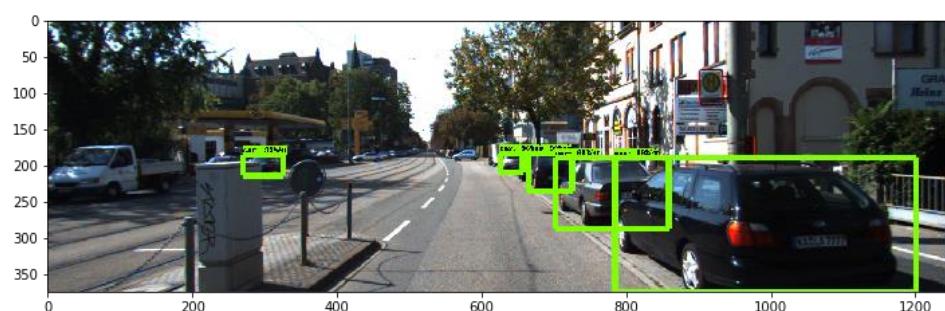


Imagen original

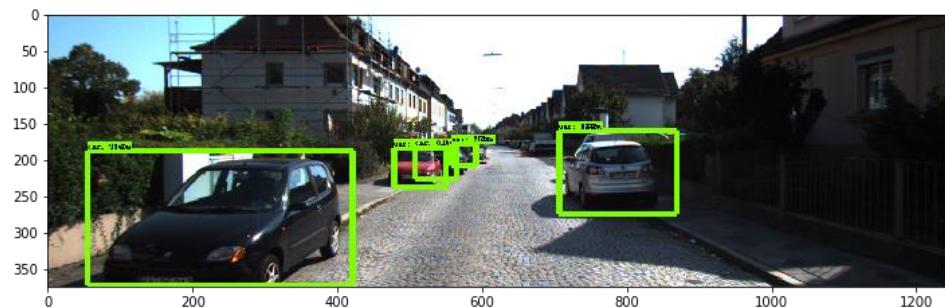


- MobileNet

-Sin reentreno

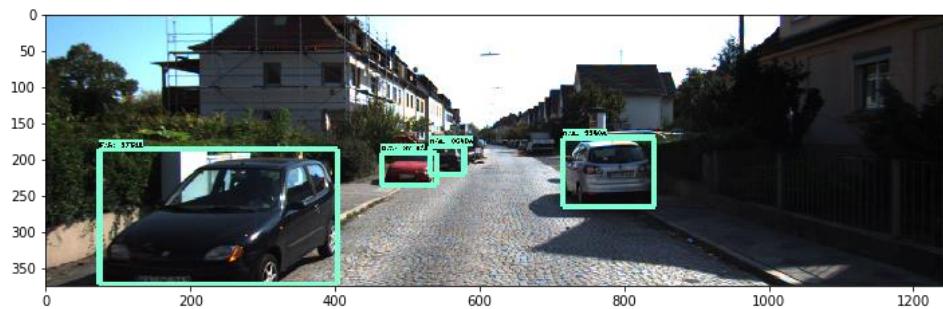


-Reentrenada



- Inception V2

-Sin reentreno



-Reentrenada



- ResNet

-Sin reentreno



-Reentrenada

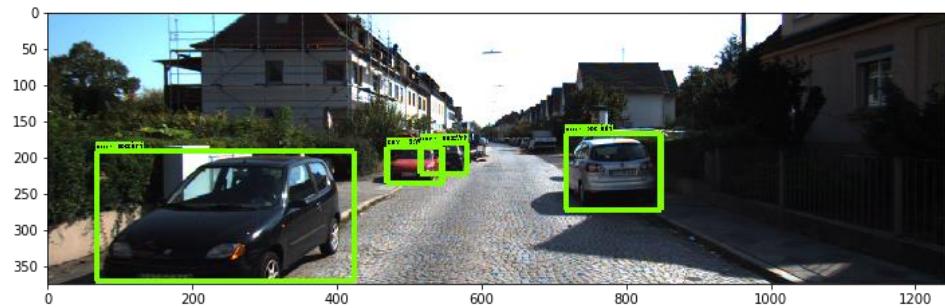
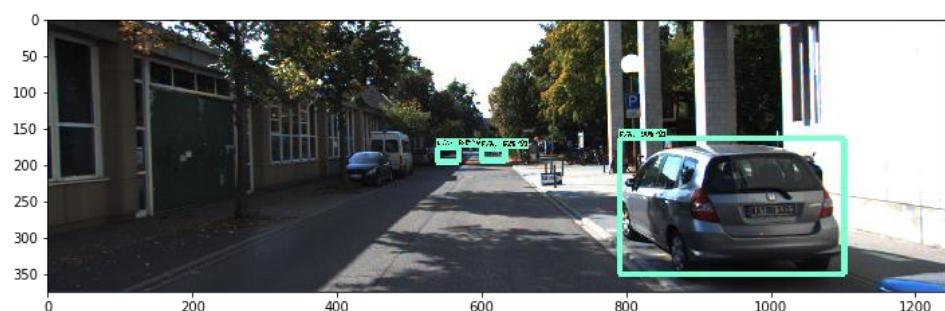


Imagen original

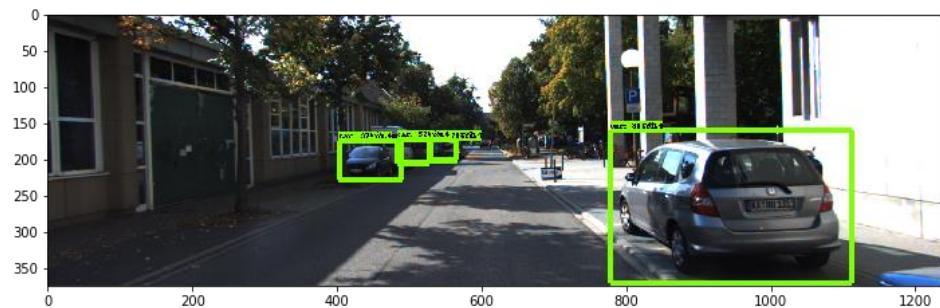


- MobileNet

-Sin reentreno

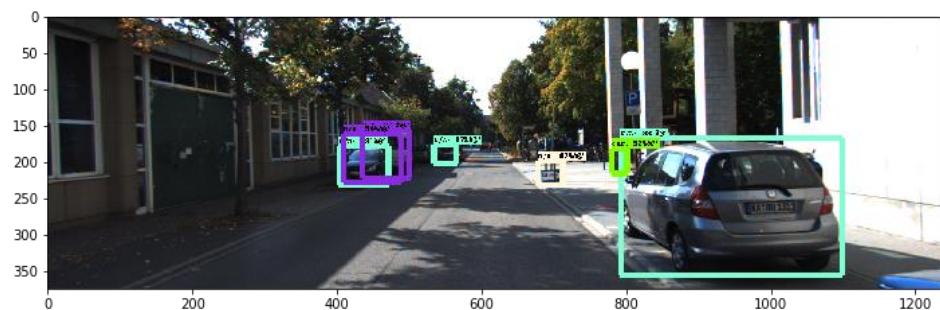


-Reentrenada

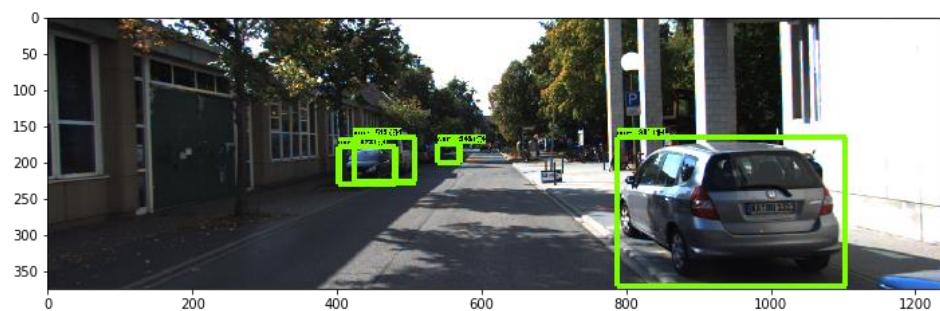


- Inception V2

-Sin reentreno

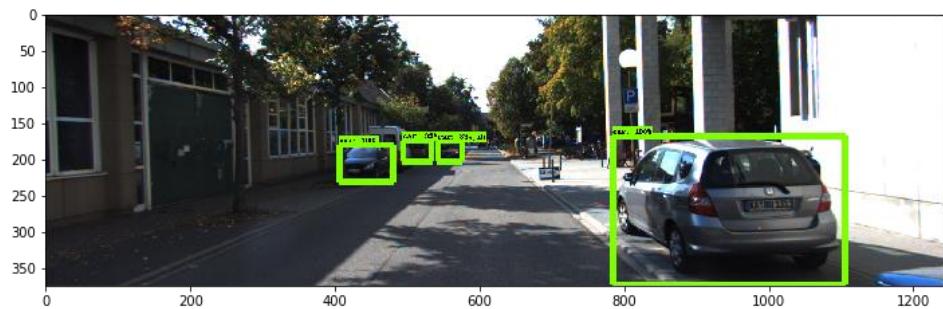


-Reentrenada:

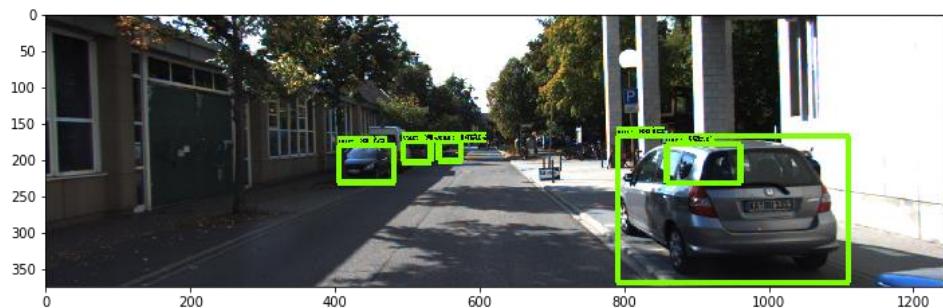


- ResNet

-Sin reentreno



-Reentrenada



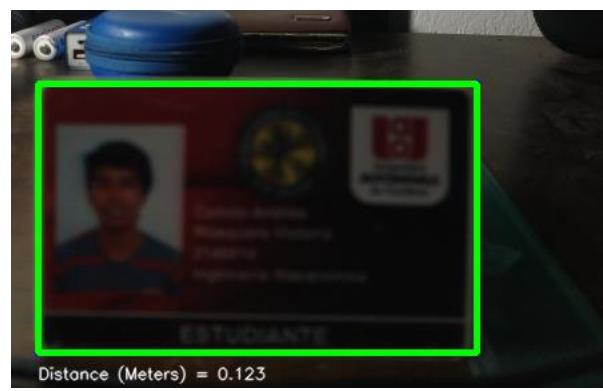
Se puede apreciar que el reentreno de la red MobileNet mejoró mucho más el desempeño de la red frente a detecciones de carros, la red Inception V2 obtuvo una pequeña mejoría y la red ResNet (la cual originalmente se entrenó con el dataset de KITTI) no presentó ninguna mejoría, de modo que su desempeño disminuyó comparado con el de la red original; esto era lo que se esperaba ya que se usaron los mismos datos tanto en el entrenamiento como en el reentreno, con la diferencia de que el entrenamiento original fue mucho más largo. Este ejercicio se realizó con la idea de evitar que la red buscara más de 1 clase, tomando en cuenta, que se quería lograr un mejor desempeño y así fuera más rápido.

Anexo B. Comprobación de distancia.



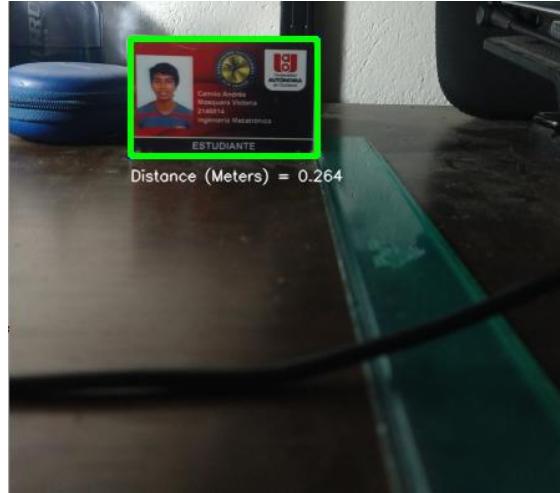
Distancia calculada = 1.814 m

Distancia real = 1.8 m



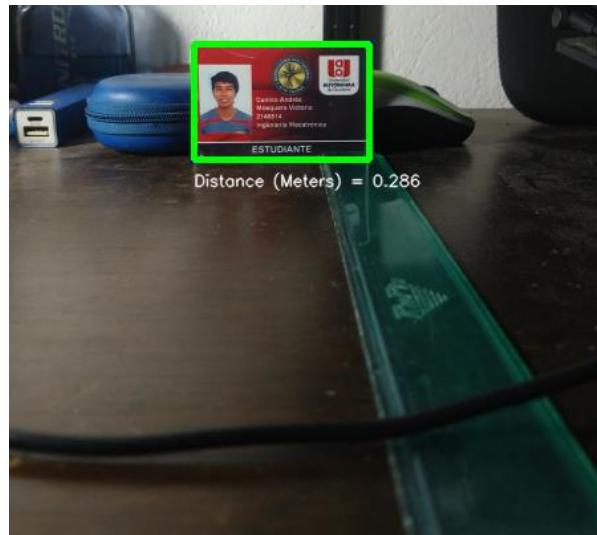
Distancia calculada = 12.3 cm

Distancia real = 12 cm



Distancia calculada = 26.4 cm

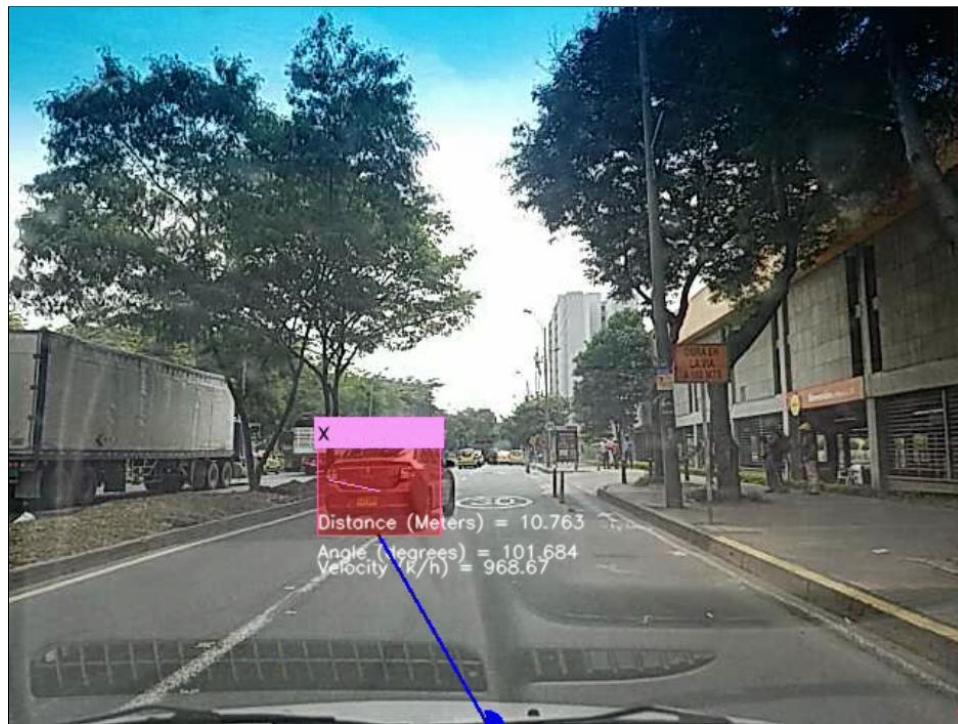
Distancia real = 27 cm



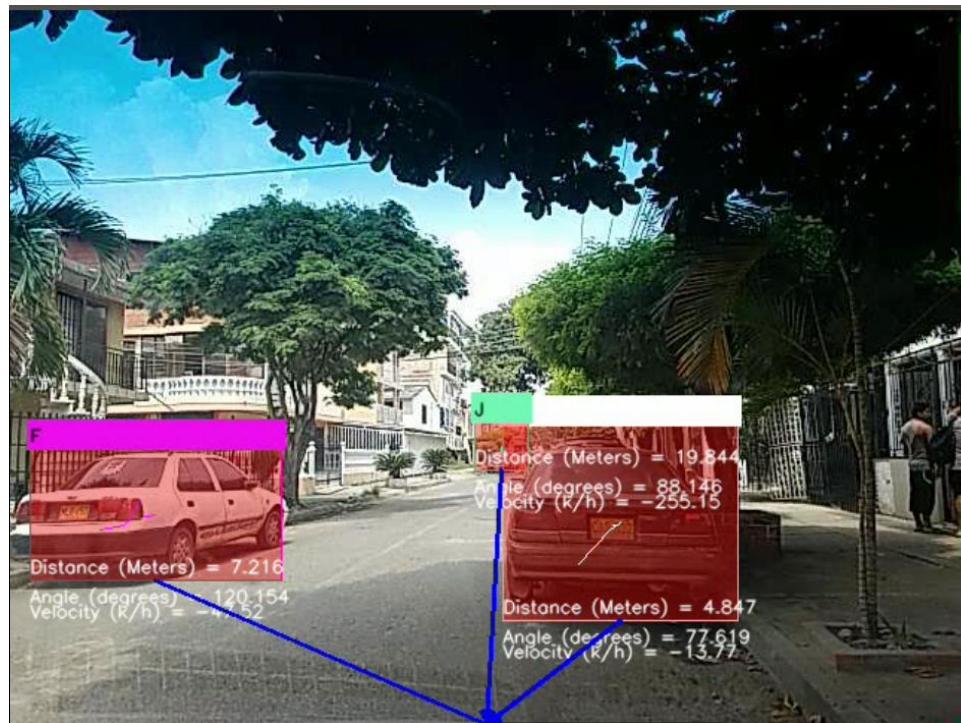
Distancia calculada = 28.6 cm

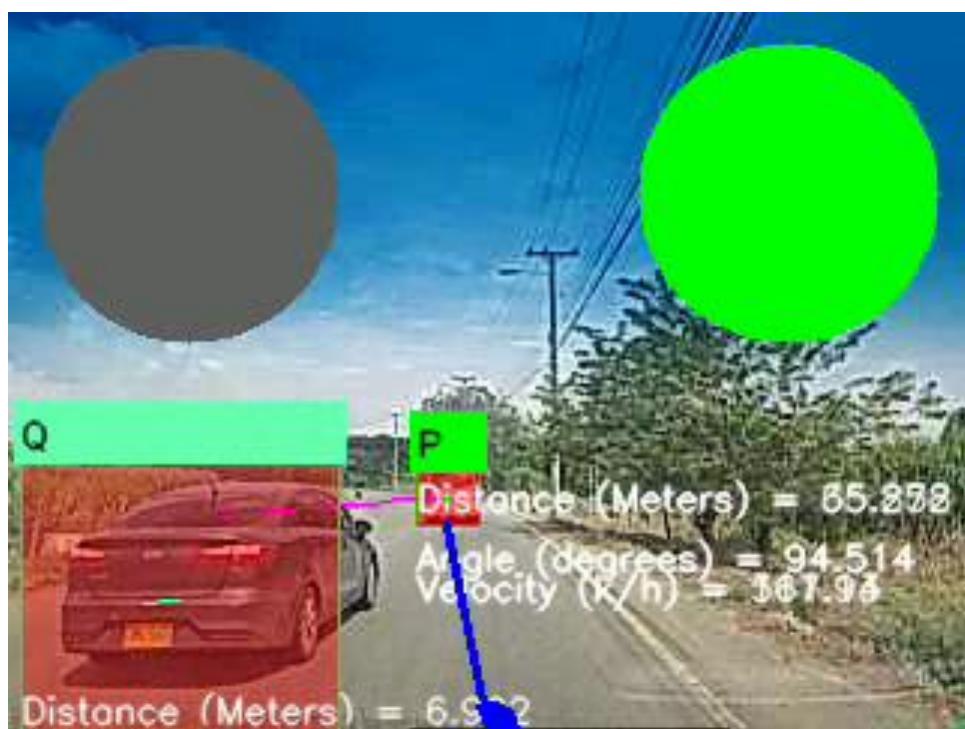
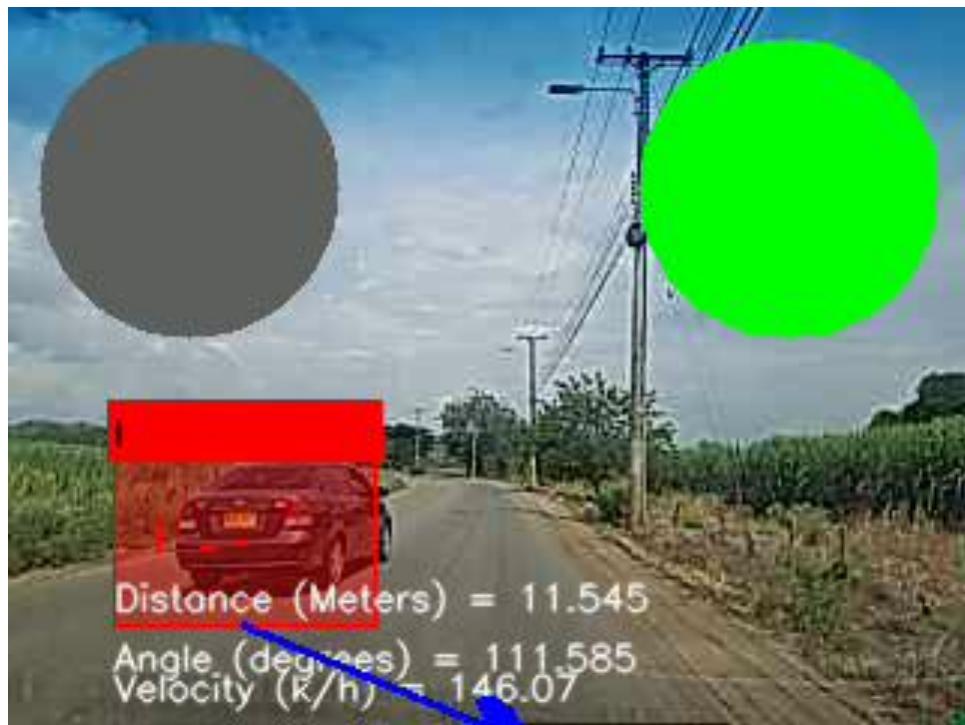
Distancia real = 30 cm

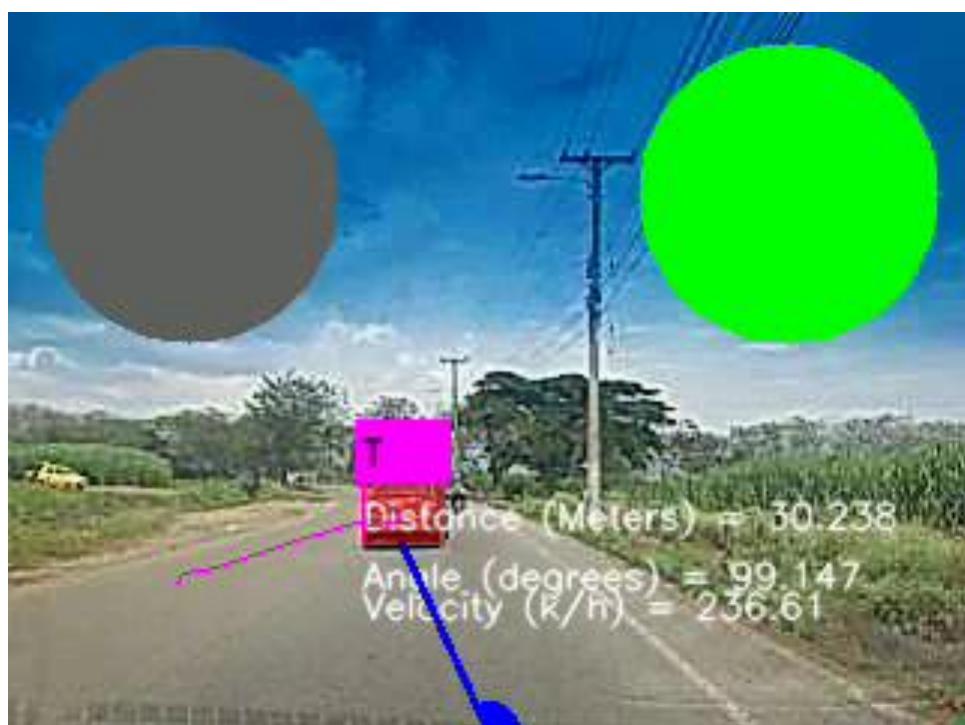
Anexo C. Validación de detección y seguimiento.

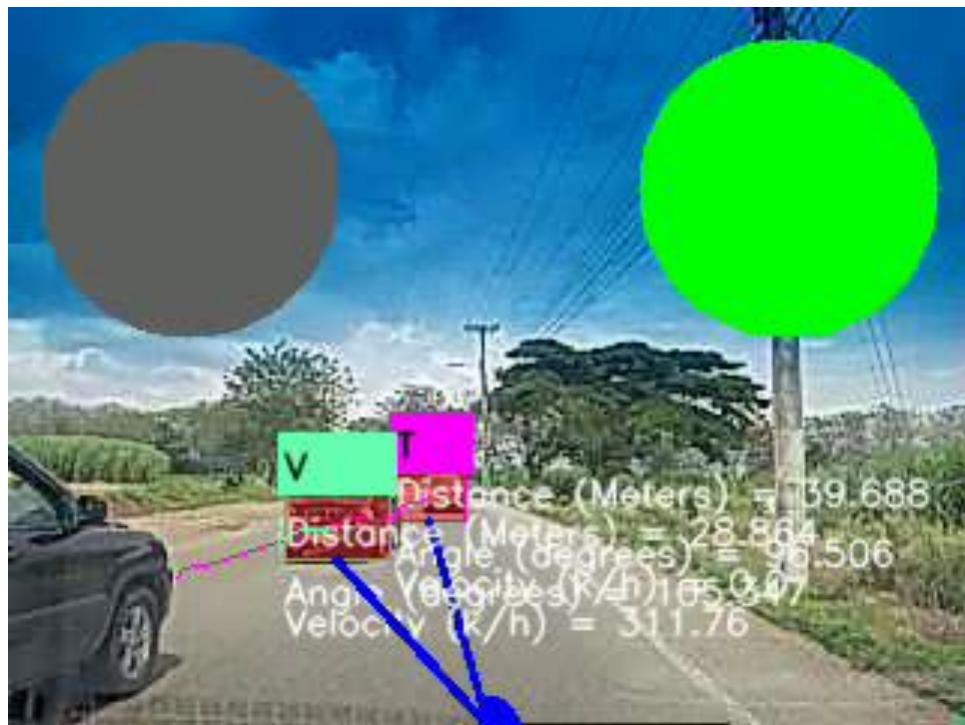


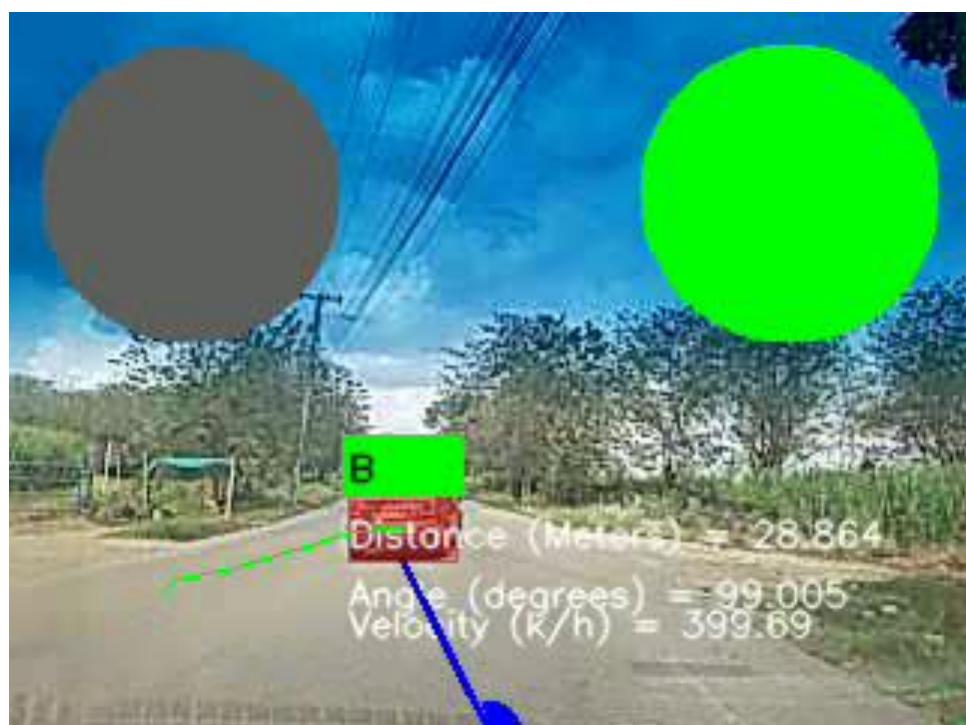
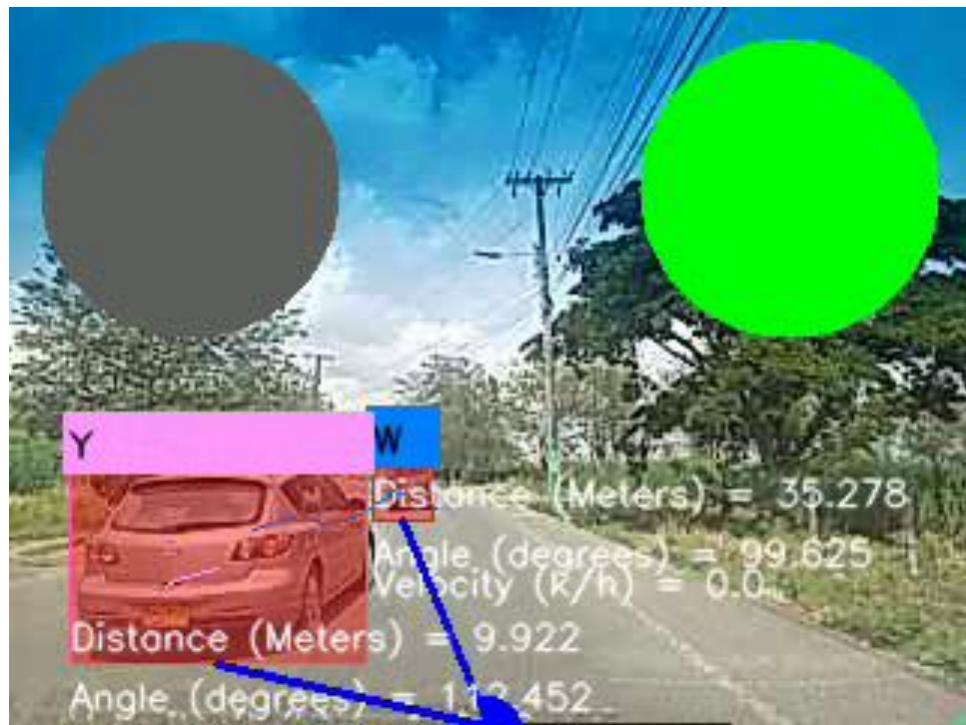












Anexo D. Filtro de Kalman.

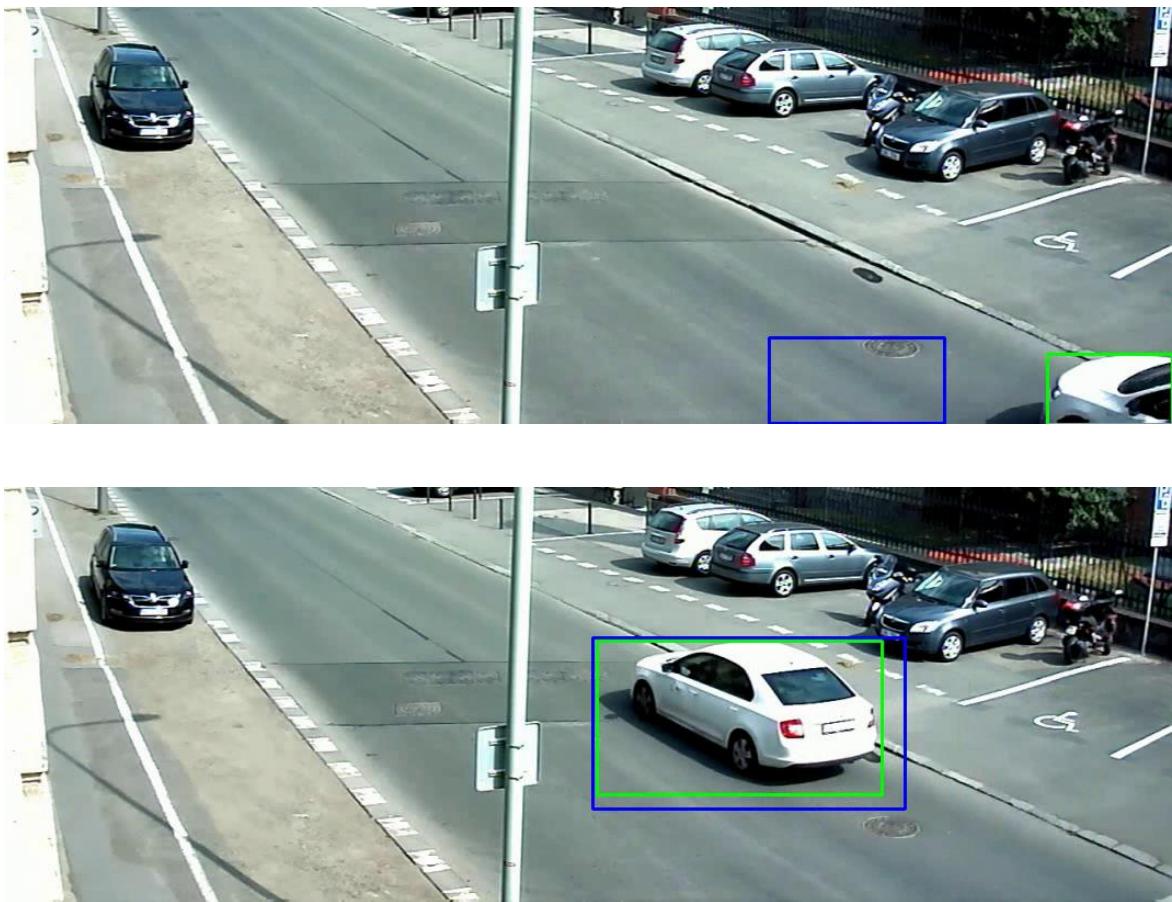
Imágenes adaptadas de [50].



En la primera imagen se aprecia en verde la detección, y en azul el filtro de Kalman. Debido a que en un principio la detección era imprecisa el filtro de Kalman se inicializa de forma incorrecta.



Poco después, el filtro de Kalman se estabiliza y empieza a seguir la detección.



Cuando la primera detección desaparece del fotograma y aparece la segunda detección, el filtro de Kalman tiene que volver a calibrarse. Esto nos permite observar que un buen filtro de Kalman no sirve de nada si no se cuenta con una buena detección, ya que al final, el filtro seguirá a la detección, por esta razón el filtro de Kalman predice la trayectoria de la detección sí esta desaparece momentáneamente, reduciendo el tiempo de recalibración sí la detección vuelve a aparecer.

Anexo E. Validación del entrenamiento con YOLO con imágenes del dataset de KITTI.

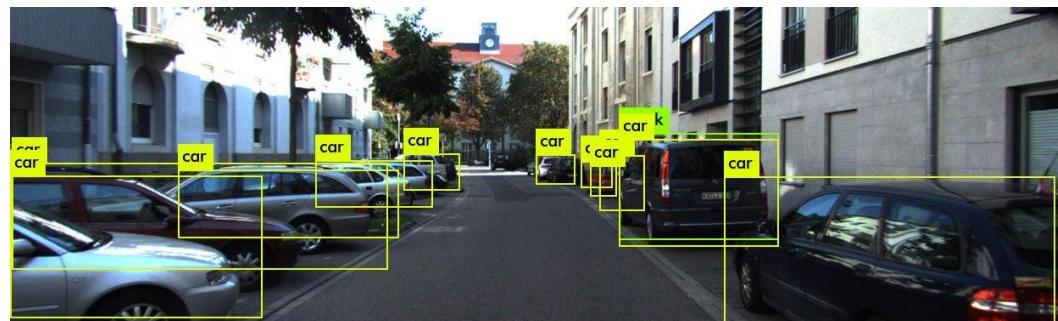
Todas las imágenes originales provienen de [22]:

Imagen Original

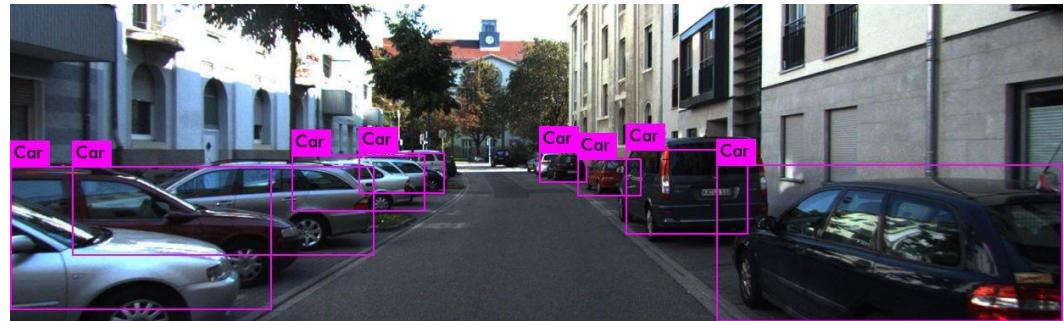


- YOLO V3

-Sin reentreno

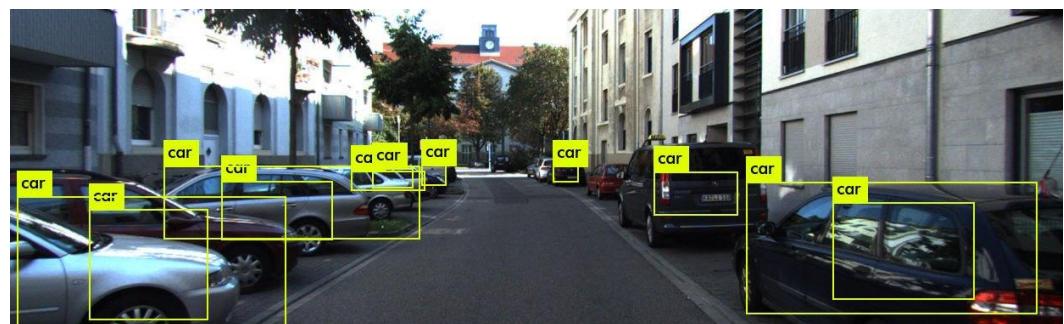


-Reentrenada



- YOLO V3 Tiny

-Sin reentreno



-Reentrenada

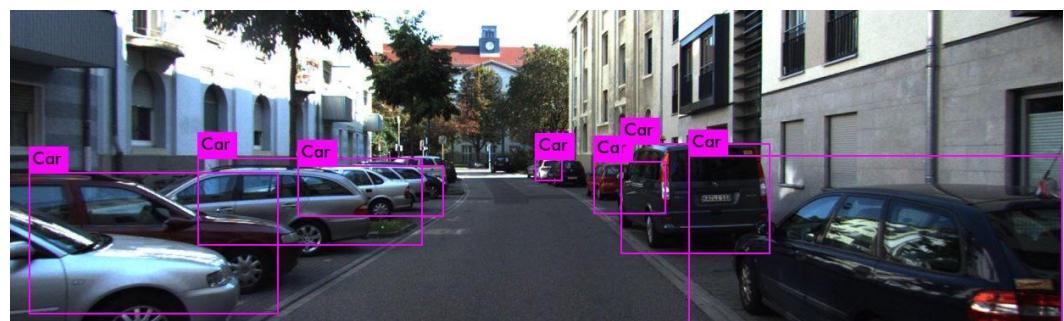
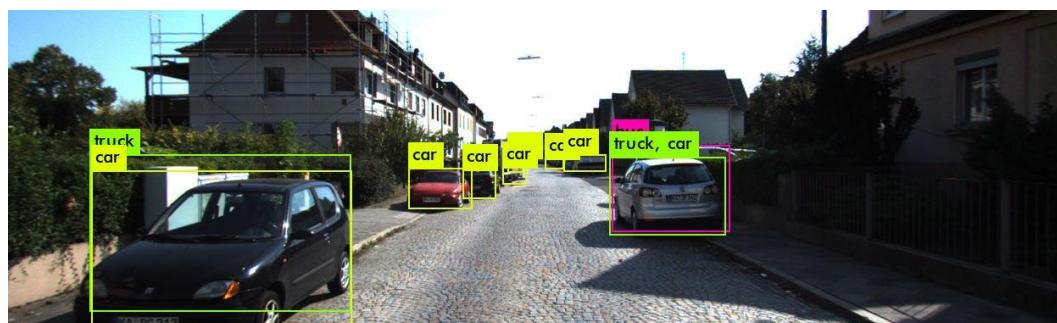


Imagen Original

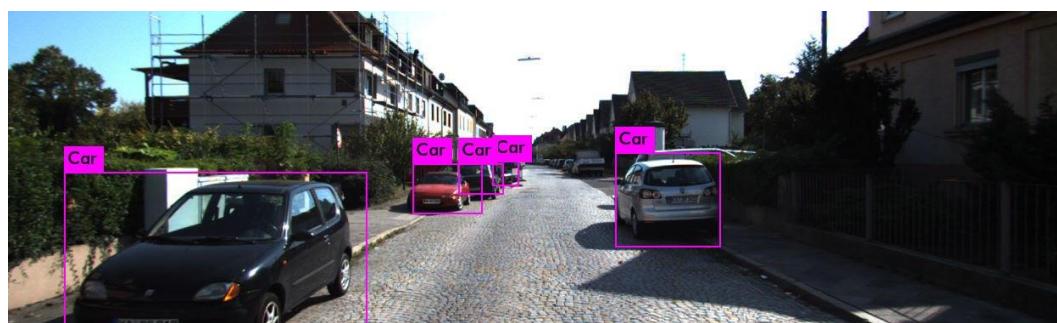


- YOLO V3

-Sin reentreno



-Reentrenada



- YOLO V3 Tiny

-Sin reentreno



-Reentrenada

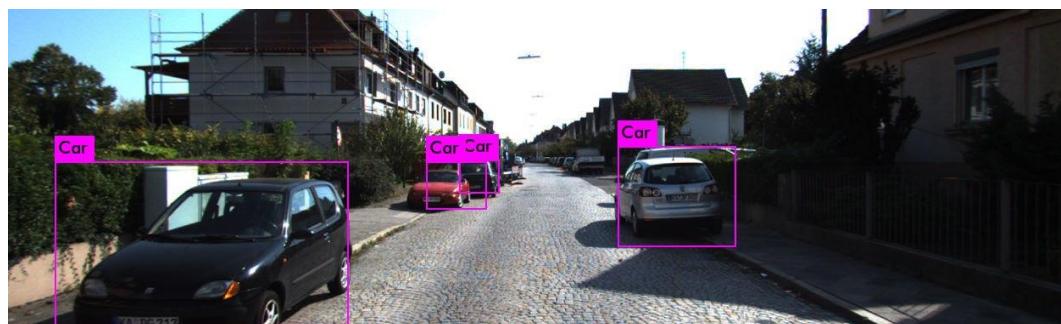
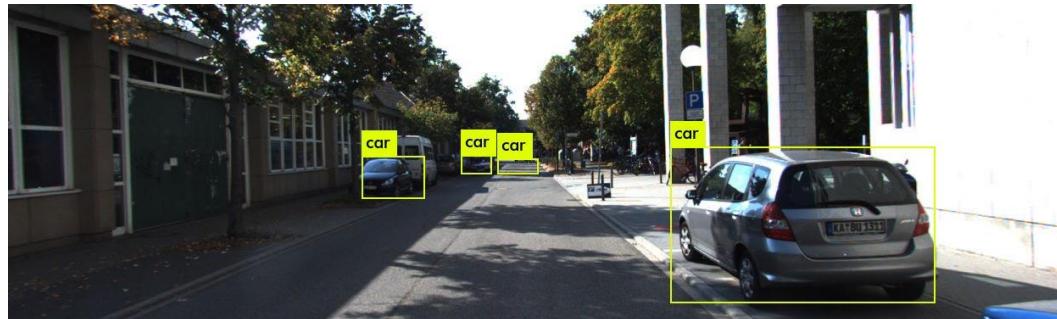


Imagen Original

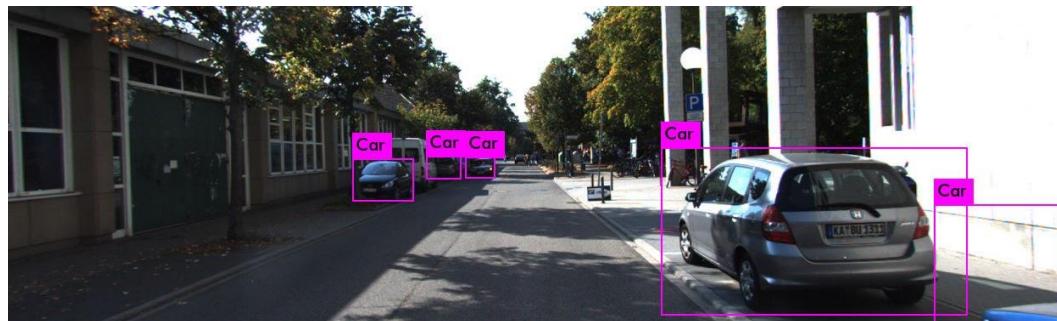


- YOLO V3

-Sin reentreno

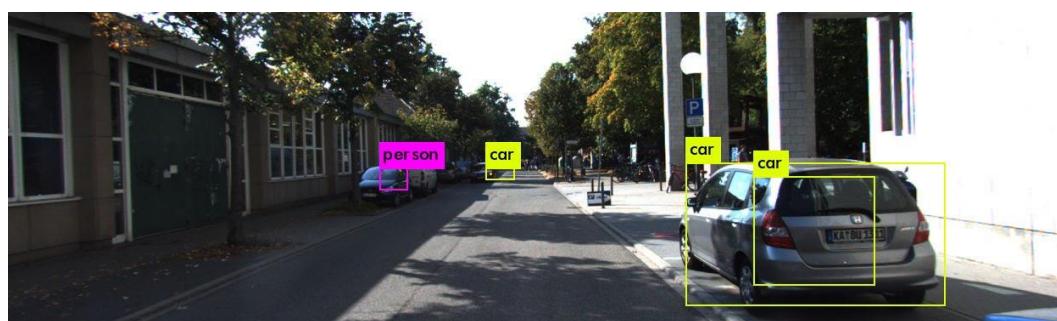


-Reentrenada

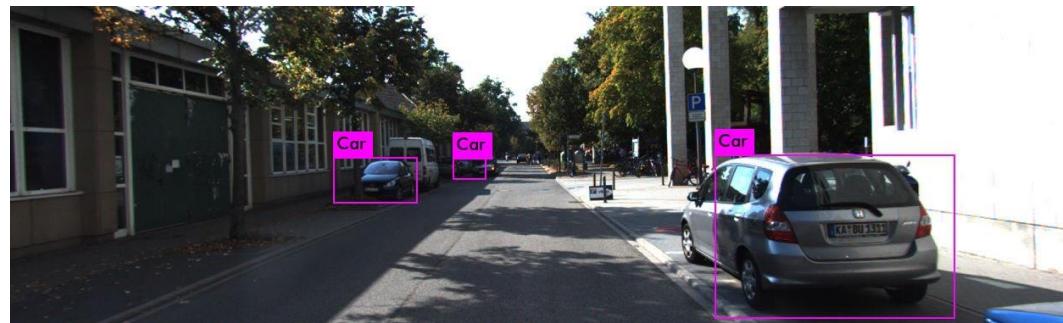


- YOLO V3 Tiny

-Sin reentreno

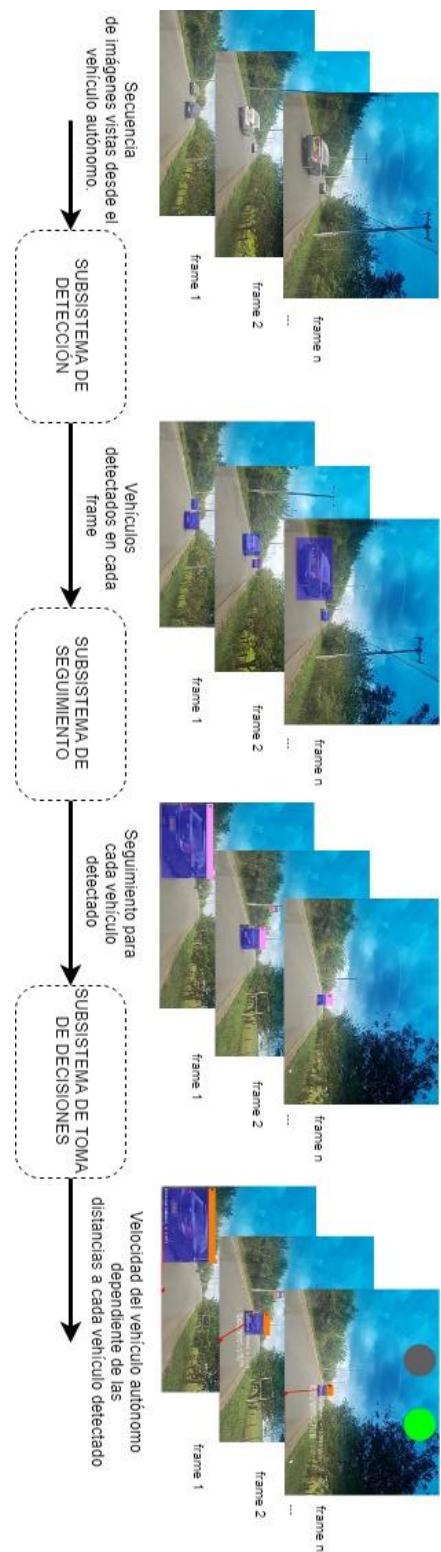


-Reentrenada

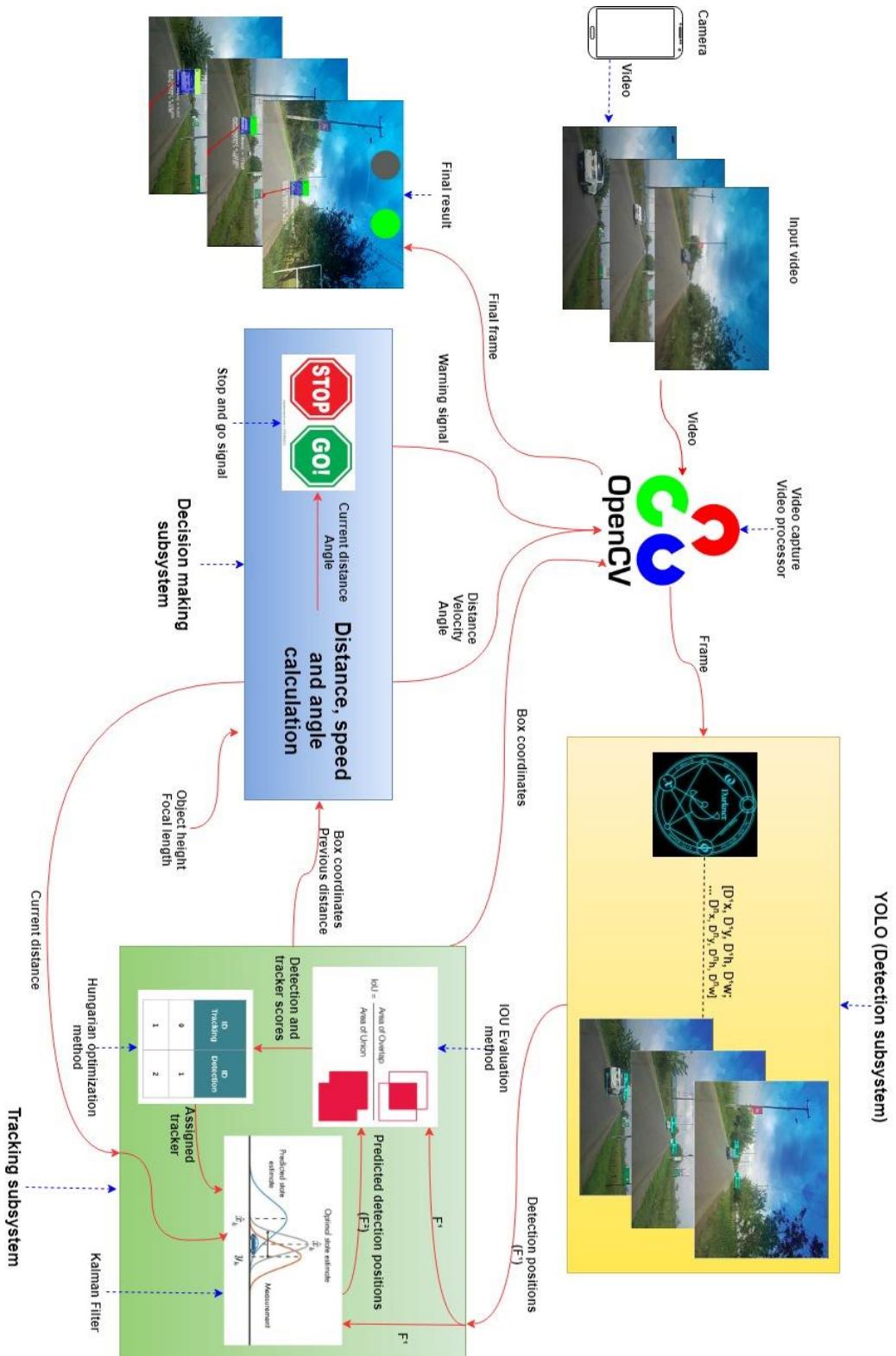


Inesperadamente se encontraron mejorías en el reentreno de ambas redes, ya que inicialmente se pensaba que se mejoraría bastante la precisión de la YOLO V3 Tiny (lo cual fue cierto), pero la YOLO V3 también obtuvo una mejoría, teniendo en cuenta que la red más pequeña (Tiny - 23 capas) se entrenó hasta 5000 pasos, mientras que la red grande (YOLO V3 - 106 capas) se entrenó hasta los 3000 pasos, debido al tiempo.

Anexo F. Diagrama general del proyecto.



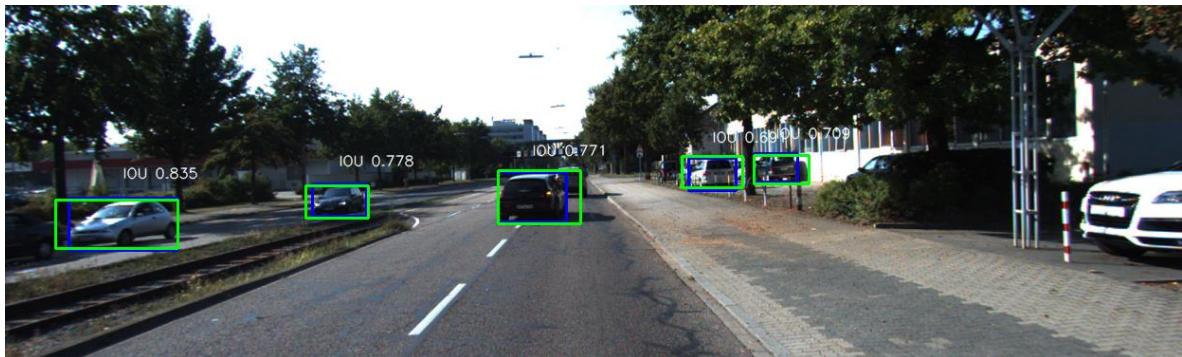
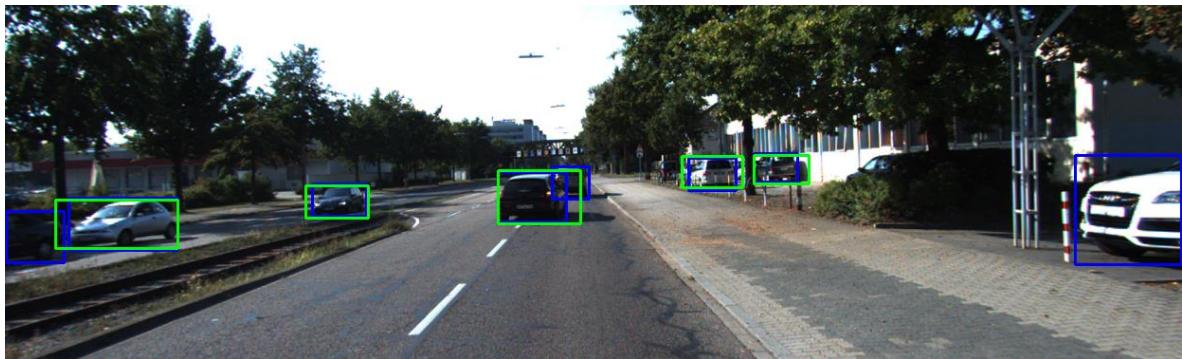
Anexo G. Diagrama del proyecto



Anexo H. Resultados de la evaluación de los detectores.

Imágenes adaptadas de [22]:

A continuación, se presenta un ejemplo de la evaluación IOU que se realizó para cada detector, en este caso, la imagen 6 fue evaluada con el detector MobileNet v2. En la primera imagen se observan las detecciones (en verde), y las etiquetas verdaderas (en azul), como se puede observar hay más etiquetas que detecciones.



En la segunda imagen se observa el resultado final de la evaluación, resaltando solo las detecciones que se pudieron relacionar con las etiquetas, este método (IOU) mira la relación que hay entre la etiqueta y la detección, por lo que entre más precisa sea esta relación mayor puntaje obtendrá. Para poder lograr los resultados mostrados fue necesario calcular el punto medio tanto de las detecciones como de las etiquetas. Por último, se calculó la distancia euclidiana entre estos puntos, la cual se tuvo en cuenta para todas las detecciones y etiquetas, por lo que se requirió implementar el método de optimización húngaro para asignar las detecciones a las etiquetas más cercanas.