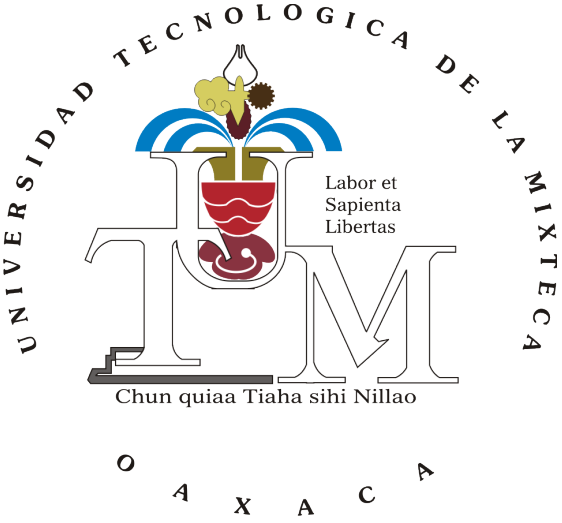


UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA

Detección de escaleras basado en aprendizaje profundo implementado en un robot de búsqueda y rescate para navegación semiautónoma

Tesis
Que para obtener el título de
Maestro en Robótica

Presenta:
José Armando Sánchez Rojas

Director de tesis:
Dr. José Aníbal Arias Aguilar

Co-Director de tesis:
Dr. Alberto Elías Petrilli Barceló

Huajuapan de León, Oaxaca, Febrero de 2022

Resumen

El objetivo principal de la robótica para desastres es salvar vidas. A causa de esto, se requieren robots capaces de realizar misiones tales como: búsqueda y rescate, reconocimiento y mapeo, eliminación de escombros e inspección estructural. Sin embargo, los errores humanos son un problema común en esta área, por lo tanto, es necesario implementar algoritmos que faciliten la teleoperabilidad y permitan que se realicen misiones sin percances. En este trabajo se presenta una propuesta para incrementar la autonomía y mejorar la teleoperabilidad en un robot de búsqueda y rescate al implementar un sistema de detección y alineación de escaleras.

La propuesta comienza con una introducción de la robótica para desastres y posteriormente se presenta el estado del arte. Con esto se sintetiza el planteamiento del problema y se justifica el trabajo a realizar. También, se incluyen los objetivos a cumplir y se plantean limitaciones y alcances del trabajo de tesis. Posteriormente, se presentan las bases teóricas necesarias para el desarrollo del trabajo. Esto incluye la disección de un robot de rescate, en sistemas, y una descripción del robot en el que se implementará el sistema propuesto.

El desarrollo del trabajo inicia con la obtención del detector de escaleras basado en aprendizaje profundo y posteriormente se utiliza el modelo para implementar el algoritmo de caracterización de las escaleras. Al obtener estas dos etapas se realiza la implementación del sistema utilizando ROS y se hacen pruebas en simulación y de manera física utilizando robots de tipo diferencial. Adicionalmente, se describe el diseño del controlador difuso necesario para el acercamiento y la alineación y se implementa el sistema completo sobre los robots.

Posteriormente, se presentan los resultados experimentales en donde se muestran las pruebas realizadas sobre el conjunto de imágenes de prueba y también las pruebas de acercamiento y alineación utilizando un robot diferencial. Por último, se incluyen las conclusiones y trabajos futuros.

A mis padres, con mucho amor.

Agradecimientos

A mis padres, Armando Sánchez y Leticia Rojas, y a mi familia por todo su apoyo, amor, sus consejos y por toda su confianza.

Al Dr. Alberto Elías Petrilli Barceló por su apoyo, sus consejos y su amistad durante el desarrollo de este proyecto.

A mi director de tesis, el Dr. José Aníbal Arias Aguilar y mis sinodales el Dr. Arturo Téllez Velázquez, el Dr. Rosebet Miranda Luna, el Dr. Eduardo Sánchez Soto y el Dr. Antonio Orantes Molina por su tiempo brindado durante esta investigación.

A Karla por su cariño y motivación que me brindó durante el desarrollo de esta tesis.

A mis amigos, compañeros y a los técnicos de la universidad que me brindaron ayuda durante el desarrollo de este trabajo.

Finalmente, al Consejo Nacional de Ciencia y Tecnología (CONACYT) y a la Universidad Tecnológica de la Mixteca por brindarme las herramientas necesarias para desarrollar este trabajo.

Índice

Índice de figuras

Capítulo 1

Introducción

En el área de la robótica de búsqueda y rescate, los desastres han sido los impulsores para el avance de esta tecnología. Los inicios de esta área fueron en 1995, después del terremoto de Hanshin-Awaki en Kobe, Japón y el bombardeo del Edificio Federal Murrah en la ciudad de Oklahoma, Estados Unidos [2]. Estos eventos despertaron el interés de diferentes investigadores e instituciones por desarrollar robots con fines de ayuda humanitaria, especialmente para zonas de alto riesgo.

La robótica para desastres se encarga de permitir que los equipos de primeros auxilios y otras partes interesadas perciban y actúen a distancia del incidente [3]. Después de una catástrofe, los robots pueden realizar tareas tales como: recolección de información, inspección estructural, búsqueda de sobrevivientes o servir como repetidores inalámbricos. Los retos para los robots terrestres surgen del ambiente desordenado y complejo donde operan, los cuales son una combinación de elementos verticales y horizontales con superficies irregulares. Por lo tanto, es muy difícil para los humanos obtener suficiente conocimiento de los alrededores del robot para guiarlo de manera óptima en entornos desconocidos. A causa de esto, la teleoperación pura no es deseable.

Uno de los obstáculos más comunes en zonas de desastres son las escaleras, por lo cual, un error humano al atravesar este obstáculo ocasionaría una misión fallida. Para mejorar la teleoperabilidad y disminuir la carga del teleoperador, en el presente trabajo de tesis se realiza la detección automá-

tica de escaleras mediante el uso de aprendizaje profundo y se utiliza un controlador que permite al robot alinearse con las escaleras de manera autónoma.

1.1. Estado del arte

La mayoría de robots para búsqueda y rescate son teleoperados e integran sistemas autónomos de bajo nivel, que buscan reducir la carga del operador para que se enfoque en otras tareas y así se facilite la misión que se pretende realizar. Para incrementar el nivel de autonomía se han presentado múltiples soluciones para ascender y descender escaleras de manera autónoma. Los algoritmos presentados se basan en tres etapas clave: *detección*, *alineación* y el *cruce* del obstáculo.

Para la detección de escaleras se han propuesto sistemas que utilizan diferentes métodos de sensado. Por ejemplo, cámaras monoculares [4], de profundidad [5] o sensores LIDAR [6]. A causa de esto, se han desarrollado distintos algoritmos, los cuales se basan en la extracción de bordes de los escalones o de planos horizontales y verticales de la huella y contrahuella de las escaleras [7].

Los métodos que se basan en la extracción de bordes utilizan principalmente cámaras RGB o RGB-D. Se han propuesto sistemas que utilizan filtros de Gabor para la extracción de texturas y eliminar los efectos de la iluminación y sombras [5], [4], [8]. Posteriormente, estos métodos buscan lineas paralelas y determinan si existe o no una escalera. Sin embargo, los algoritmos consideran que los bordes detectados en la imagen son horizontales o casi horizontales (pendiente menor a 0.5) para su funcionamiento. También, se han considerado casos en donde se busca un sistema simple y se extraen bordes utilizando filtros Gaussianos y el operador Canny y posteriormente una transformada de Hough para detectar las lineas que son aproximadamente horizontales [9].

En [10], se presenta una detección de lineas, las cuales representan discontinuidades que existen en las imágenes RGB-D obtenidas de una cámara Kinect. Las discontinuidades surgen debido al ángulo en que se observan las escaleras cuando el sensor se posiciona para que se observe solamente la contrahuella de los escalones. En [11], se presenta un algoritmo similar; sin embargo,

se sitúa la cámara de tal manera que se puedan detectar los bordes que definen la huella de la escalera.

También se han propuesto métodos que utilizan una combinación de imágenes RGB y RGB-D para la detección [12]. Al hacer esto, se evitan algunos problemas que surgen durante la medición de distancias bajo luz solar, por los sensores que utilizan un patrón de rayos infrarrojos para medir la distancia. Este sistema determina si los pixeles de profundidad son válidos, de lo contrario utiliza solamente las imágenes RGB. Para la detección de bordes utilizan un filtro de Sobel sobre la imagen RGB-D y una transformada de Hough para detectar bordes de las imágenes RGB.

Algunos sistemas descritos anteriormente tienen falsos positivos, debido a los ambientes desordenados donde operan los robots, especialmente cuando los bordes de las escaleras no se encuentran en posición horizontal en la imagen. A causa de esto, se han propuesto trabajos que utilizan sensores LIDARs para realizar la detección de las escaleras [13], [14]. Algunos posicionan un LIDAR de manera vertical (ver Figura 1.1), para determinar la forma de las escaleras; y además consideran que las escaleras se encuentran en el rango del LIDAR. En la Figura 1.1, se muestra cómo se realiza el escaneo durante este tipo de detección [13]. Por otra parte, [6] presenta un algoritmo de detección de escaleras utilizando un LIDAR, obteniendo planos horizontales y verticales de las escaleras.

Mihankhah [15] presenta un sistema de detección y un control difuso para subir escaleras de manera autónoma. Las entradas del control son los datos de dos LIDARs, uno que escanea de manera horizontal (HLRV, por sus siglas en inglés) y otro de manera vertical (VLRF, por sus siglas en inglés). La detección se hace con la medición de los escalones a partir de la nube de puntos del VLRF, sin embargo este algoritmo no es robusto y el campo de visión es menor al de una cámara. Para confirmar la existencia de las escaleras se verifica que el LRFH detecte la cara frontal de un escalón entre los 45 y 135 grados del sensor. Después de la detección, utilizan un control difuso para mantener al robot paralelo a la pared derecha. Sin embargo, esto presenta un problema para la navegación cuando no existen paredes.

Para el cruce de las escaleras de manera autónoma, [16] presenta un algoritmo implementado un

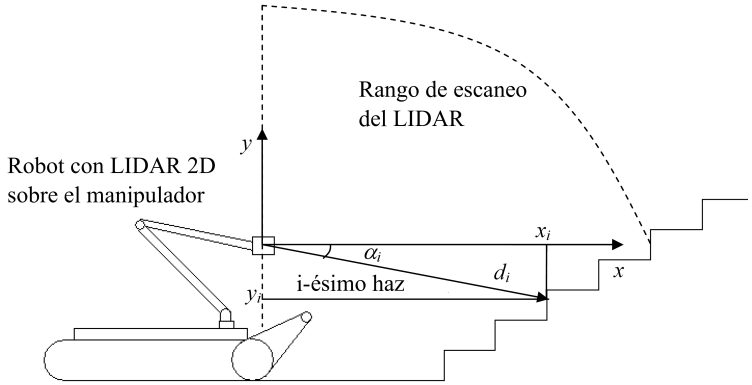


Figura 1.1. Escaneo del LIDAR para detección de escaleras [13].

robot con orugas (ver Figura 1.2). El algoritmo se basa en el uso de un filtro de Kalman extendido (EKF, por sus siglas en inglés), para la estimación y fusión de mediciones de un giroscopio de tres ejes y los bordes obtenidos con una cámara abordo. El filtro permite estimar la orientación del robot durante su trayectoria y la estimación se actualiza utilizando las mediciones de la cámara. La solución que se propone es un método seguro, rápido y robusto, para subir escaleras de manera autónoma; sin embargo no presenta una detección de las escaleras.

Además, para el funcionamiento del algoritmo, se asume que las escaleras son rectas y que los bordes son paralelos. A causa de esto, la solución no funcionaría en escaleras helicoidales. La detección de los bordes se tarda aproximadamente 60 ms y antes de iniciar el cruce, el robot debe permanecer estático durante 5 s para inicializar su vector de estados y su covarianza, lo que causa un retraso considerable cuando existe un número significativo de pisos para explorar. Por otra parte, en menos de 10 % de las pruebas, el ancho del camino no se detectaba y ocasionaba que el robot colisionará con los barandales o las paredes.

Por otro lado, la mayoría de robots con orugas tienen aletas que permiten atravesar obstáculos más grandes que el tamaño de sus orugas principales. Sin embargo, las aletas incrementan la dificultad para controlar al robot y la carga de trabajo del operador. Para eliminar estos inconvenientes, se presenta un algoritmo de aletas activas [17], las cuales se mueven de manera autónoma, cuando el robot atraviesa obstáculos o escombros. Para el control se utilizan dos LIDARs, uno en cada lado del robot, para obtener el perfil de la superficie que atraviesa (ver Figura 1.3). Esta propuesta

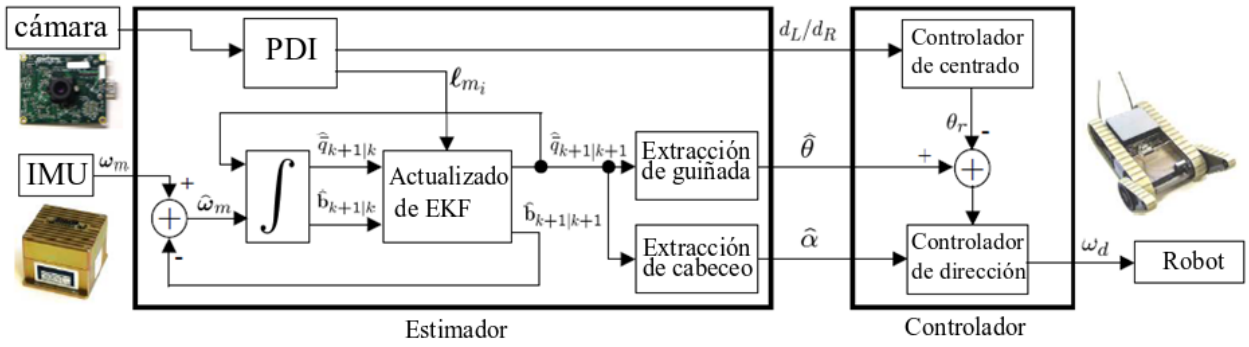


Figura 1.2. Filtro de Kalman Extendido para la estimación de orientación [16].

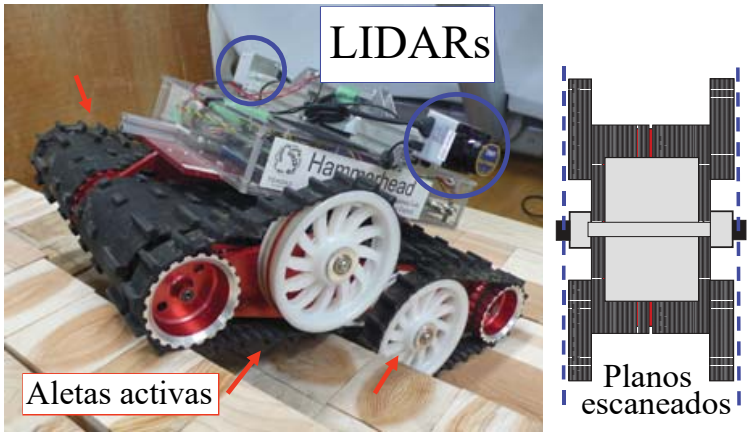


Figura 1.3. Posicionamiento de los LIDARs para las aletas activas [17].

es efectiva y robusta, pero, a causa de los sensores, el costo del prototipo es muy alto.

A pesar de que el problema de ascender escaleras ha sido el más estudiado, para descenderlas es lo contrario. En [18], se presenta un algoritmo para la detección, alineación y descenso del robot sobre las escaleras. El sistema utiliza una cámara monocular y la textura de la imagen para detectar cambios de profundidad y así encontrar el inicio del obstáculo. Para mejorar su algoritmo; proponen utilizar una nube de puntos, lo que permitiría detectar y navegar con menor dificultad. En [19] y [20], se han propuesto algunos métodos que utilizan nubes de puntos obtenidas de sensores RGB-D. Estos algoritmos realizan un análisis estructural del obstáculo para determinar las características que los definen (principalmente bordes) y posteriormente realizan una representación en 3D de las escaleras.

Otro método de detección de escaleras utiliza redes neuronales convolucionales (CNNs, por sus

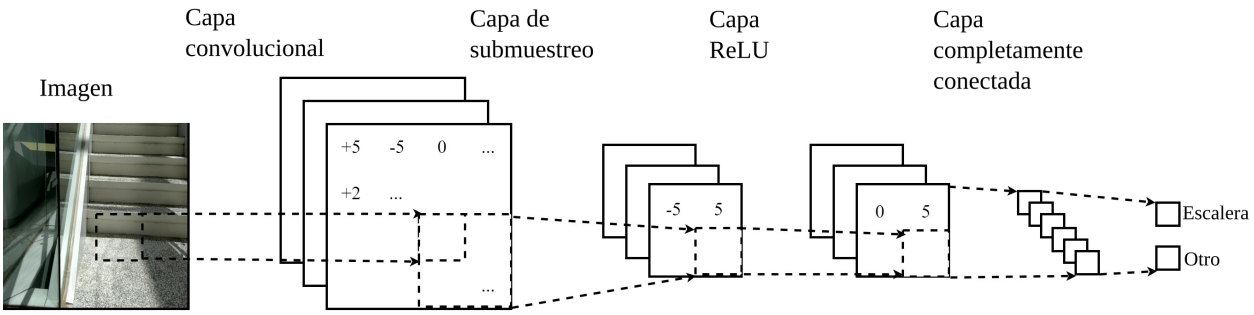


Figura 1.4. Estructura de la red neuronal convolucional para detección de escaleras [21].

siglas en inglés) para la detección de escaleras. Una de las ventajas de utilizar este método es que puede ser entrenado para detectar diferentes tipos de escaleras, incluyendo espirales y curvas. De igual manera, debido a la escalabilidad de las redes neuronales, el algoritmo se puede entrenar para detectar otros tipos de obstáculos. La estructura de las CNNs es la base para la detección de objetos y clasificación de imágenes. Por ejemplo, en [21] se utiliza una CNN de cuatro capas: convolucional, de agrupamiento, de ReLU (Unidad Rectificada Lineal) y una capa completamente conectada. Esta estructura fue entrenada con escaleras tradicionales, aunque el autor propone utilizar otros tipos de escaleras, lo que incrementaría la percepción del robot autónomo.

1.2. Planteamiento del problema

Los robots de búsqueda y rescate tienen que desplazarse en ambientes complejos y realizar tareas en las zonas afectadas. Estos ambientes incluyen obstáculos tales como escaleras o rampas, lo cual dificulta la teleoperación y, por lo tanto, el grado de movilidad depende directamente de la habilidad del operador. A causa de esto, el operador no se enfoca en las tareas propias de búsqueda y rescate, lo que incrementa el tiempo necesario para realizar las misiones.

Para mejorar la teleoperabilidad, se han realizado múltiples trabajos de investigación que buscan disminuir la carga del operador al implementar sistemas de detección de obstáculos. Para la detección de escaleras, se han propuesto sistemas basados en extracción de bordes y/o planos. Sin embargo, estos algoritmos tienden a fallar cuando existen más elementos en la imagen o se pre-

senta un ambiente desordenado. También, para estos algoritmos, las escaleras tienen que ser casi horizontales dentro de la imagen, de lo contrario la solución no funcionaría.

Al implementar la detección de escaleras utilizando aprendizaje profundo, se busca incrementar la precisión y evitar los problemas mencionados anteriormente. Sin embargo, la utilización de aprendizaje profundo también implica ciertas complicaciones que incluyen, la recolección de bases de datos y el diseño de la arquitectura de redes neuronales para lograr la clasificación del obstáculo.

1.3. Justificación

En zonas de desastres el tiempo es un factor crítico para salvar vidas. Los robots de búsqueda y rescate deben de moverse lo suficientemente rápido para llegar a todos los sobrevivientes potenciales, pero lo suficientemente lento para evitar crear colapsos adicionales [22]. Sin embargo, la teleoperación es una tarea difícil para operadores con poca experiencia y los errores humanos son la causa principal de una misión fallida [15]. Subir escaleras es una tarea difícil debido al nivel de percepción del operador y a causa de los derrapes que pueden surgir durante el movimiento. Esto causa una desviación inesperada para el operador y podría causar que el robot caiga o se atore. Al eliminar el factor humano durante el movimiento en escaleras, el robot tendrá una mejor probabilidad de subir sin percances. Además, se eliminarían fallas al interpretar información recibida de los sensores del robot.

Por otra parte, con la implementación de un sistema de detección de escaleras se implementará un algoritmo que permita al robot alinearse con las escaleras de manera autónoma. Al incrementar la autonomía del robot se facilitaría la teleoperación en entornos desconocidos y permitiría que el operador se enfoque en tareas propias de búsqueda y rescate. También, el uso de un robot semiautónomo disminuiría el tiempo requerido para la inspección de zonas afectadas. Esto debido a la facilidad de operación y la eficiencia del robot para atravesar obstáculos de manera autónoma.

En comparación con los algoritmos presentados en el estado del arte, el que se desarrollará tendrá alta escalabilidad, por lo tanto, se podrá entrenar para detectar otros objetivos, tales co-

mo obstáculos o victimas en zonas de desastre. Además, algunos algoritmos dependen de ciertas condiciones tales como el ángulo en el que se observan las escaleras, o bien, la forma de las mismas (recta o en espiral); al utilizar aprendizaje profundo se pretende eliminar estas limitaciones de los detectores actuales. Finalmente, debido a que el algoritmo se implementará en ROS (Robot Operating System), el sistema podrá ser utilizado en otros robots móviles.

1.4. Hipótesis

Utilizando aprendizaje profundo se logrará incrementar la autonomía de un robot de búsqueda y rescate al permitir que detecte escaleras y se alinee con ellas de manera autónoma.

1.5. Objetivos

A continuación se presenta el objetivo general de esta tesis y los objetivos específicos necesarios para realizar el trabajo de investigación.

1.5.1. Objetivo General

Implementar un sistema de percepción basado en aprendizaje profundo en un robot de búsqueda y rescate que permita la detección de escaleras durante el recorrido del robot.

1.5.2. Objetivos Específicos

- Desarrollar una arquitectura de redes neuronales profundas para la detección de escaleras.
- Adquirir y etiquetar las bases de datos necesarias para el entrenamiento.
- Entrenar, implementar y optimizar la arquitectura de aprendizaje profundo para lograr la detección de escaleras.

- Implementar un control que permita al robot alinearse, de manera autónoma, con las escaleras.
- Realizar pruebas físicas para validar el funcionamiento del sistema.

1.6. Limitaciones

El algoritmo para la detección de escaleras se implementará en un robot de búsqueda y rescate desarrollado en la Universidad Tecnológica de la Mixteca. Esto presenta algunas limitaciones para la investigación:

- A causa de la complejidad del sistema electrónico, los sensores que se decida utilizar tendrán que adaptarse a las especificaciones del prototipo desarrollado.

Además, a causa de la complejidad con la que operan los robots de búsqueda y rescate se presentan limitaciones ambientales:

- El rango de iluminación en zonas de desastres es muy amplio, por lo tanto, se considerará una iluminación artificial en ambientes interiores.

1.7. Alcances

Con el desarrollo de este trabajo se pretende obtener un robot de rescate capaz de detectar escaleras y alinearse con ellas en una zona de pruebas. El sistema a desarrollar utilizará sensores de bajo costo tales como LIDAR 2D y/o cámara RGB-D para la detección y alineación, y se implementará en ROS para incorporarlo como paquete adicional a los desarrollados previamente.

1.8. Estructura de la tesis

Este trabajo de tesis está organizado en 5 capítulos, a continuación se exponen las ideas principales de cada uno:

Capitulo 2. En este capitulo se presentan las bases teóricas necesarias para el desarrollo de este trabajo. Aquí también incluye una descripción de la Robótica para Desastres y se definen conceptos usados a lo largo del trabajo.

Capitulo 3. En este capitulo se describe el desarrollo del sistema implementado, desde la etapa de obtención de datos hasta la implementación del sistema sobre un robot diferencial.

Capitulo 4. Este capitulo incluye los resultados obtenidos al implementar el *pipeline* sobre un robot móvil. Estos resultados muestran la detección y caracterización del obstáculo para permitir que el robot se acerque y se alinee de manera autónoma.

Capitulo 5. Se presentan las conclusiones del trabajo y propuestas de trabajos futuros para mejorar el sistema propuesto.

Capítulo 2

Marco teórico

En esta sección se presentan los fundamentos teóricos que permitirán desarrollar el sistema de detección de escaleras. Se inicia con una descripción del área de la robótica para desastres en donde se presentan las tareas más comunes de los robots de rescate y los componentes principales de estos sistemas. Posteriormente, se incluye una descripción del robot que se utilizará para implementar el sistema de detección. Finalmente, se incluyen las bases teóricas sobre la visión por computadora, aprendizaje automático (ML, por sus siglas en inglés) y los sistemas difusos.

2.1. Robótica para desastres

La robótica de rescate se dedica a permitir que equipos de respuesta y otras partes interesadas perciban y actúen a distancia del lugar de un desastres o incidente extremo. El impacto que tienen los desastres se encuentra en aumento a causa del incremento en el tamaño de las zonas urbanas y la población mundial. Por lo tanto, se ha incrementado la necesidad de robots para las distintas fases de los desastres. A esta consideración más amplia del uso de robots en todas las fases de una catástrofe se le conoce como *Robótica para desastres* [3].

2.1.1. Tareas de los robots de rescate

Un robot de rescate puede realizar distintas tareas durante todas las fases de una catástrofe. Estas tareas dependen del diseño del robot y las capacidades del mismo. Cabe mencionar que el objetivo principal de estos robots no es reemplazar a los equipos de respuesta, sino aumentar y extender sus capacidades [3].

Búsqueda es una actividad concentrada en el interior de una estructura, cuevas o túneles y el objetivo principal de esta tarea es encontrar victimas o peligros sin arriesgar a los rescatistas.

Reconocimiento y mapeo es una actividad más amplia que búsqueda, la cual permite que los equipos de respuesta tengan conciencia situacional a distancia.

Eliminar escombros para la extracción de victimas o para limpieza antes de la reconstrucción de la zona. El objetivo es mover escombros más rápido que un ser humano, pero con una menor huella que utilizando maquinaria pesada.

Inspección estructural de edificios dañados. Esta tarea tiene como objetivo proveer una mejor vista de los daños, prevenir colapsos adicionales y determinar si es posible la entrada a estructuras dañadas.

Evaluación medica e intervención in situ se necesita para que los doctores y paramédicos interactuen con las victimas y las inspeccionen visualmente o apliquen sensores de diagnostico.

Extracción y evacuación de victimas es necesaria cuando existe gente en la zona de desastre. El objetivo es mover a los sobrevivientes rápido y de manera segura a un punto de reunión.

Servir como un beacon móvil o repetidor inalámbrico y extender el rango comunicación y ancho de banda entre robots y equipos de respuesta.

2.1.2. Componentes de un robot terrestre de búsqueda y rescate

Los robots de rescate terrestres incorporan distintos elementos de software y hardware para su funcionamiento. Dentro del área de robótica para desastres, un robot móvil necesita tener cierto grado de autonomía para facilitar la teleoperación e incrementar la probabilidad de éxito en misio-

nes de búsqueda y rescate. En la Figura 2.1 se muestran los sistemas principales de un robot para búsqueda y rescate [23]. A continuación se describe cada sistema:

- Locomoción: permite la movilidad del robot a través de terrenos con superficies variadas.
- Comunicaciones: permite que haya comunicación entre el robot y el operador para informar sobre peligros o víctimas.
- Batería o fuente de alimentación: alimenta a los componentes electrónicos.
- Controlador: regula al resto de los sistemas y los mantiene trabajando juntos.
- Sensores: son utilizados para navegar, detectar peligros y/o objetivos.
- Manipulador: permite que el robot mueva obstáculos, como puertas o escombros, o llevar material de primeros auxilios a víctimas.
- Percepción: utiliza los datos de los sensores para permitir al robot estar *consciente* de sus alrededores.
- Conocimiento: engloba la habilidad del robot para modelar el mundo utilizando información a priori e información recién adquirida.
- Planeación: utiliza el conocimiento y los componentes de sensado para generar comportamientos como evasión de obstáculos, toma de decisiones y/o búsqueda de rutas alternas más rápidas y/o seguras.
- Autonomía: permite que el robot colabore con el humano/operador de manera eficiente y eficaz.
- Colaboración: determina las capacidades del robot para colaborar con otros robots.
- Interfaz humano-máquina: Permite que el operador tenga una conciencia situacional remota a través del robot.

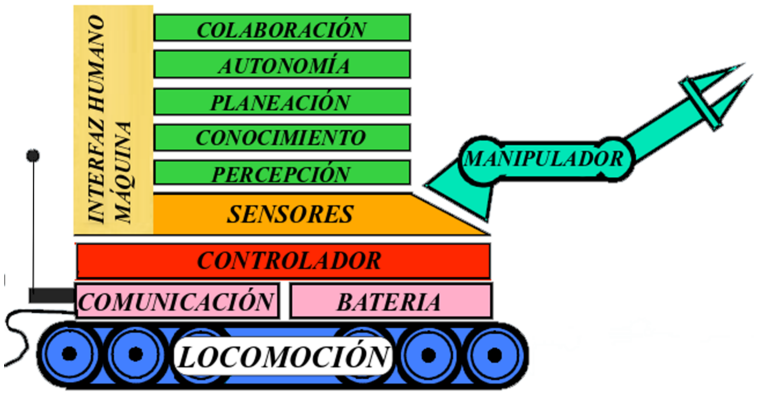


Figura 2.1. Sistemas de un robot terrestre de rescate [23].

2.2. Descripción del robot a utilizar

El robot que se utilizará para el desarrollo de esta tesis es un prototipo fabricado en la Universidad Tecnológica de la Mixteca. Este prototipo es un robot móvil de tipo diferencial, especialmente fabricado para ser utilizado en la investigación dentro del área de la robótica para desastres.

El sistema mecánico consiste de un chasis de aluminio, un sistema de suspensión y un sistema de tracción que utiliza un par de orugas principales y un par de orugas auxiliares, mejor conocidas como aletas (ver Figura 2.2) [24]. El sistema mecánico está diseñado para trabajar en ambientes de desastre, donde existen superficies irregulares con obstáculos tales como rampas y escaleras.

La interfaz electrónica del robot consiste de tres subsistemas principales: de alimentación eléctrica, de control y de sensado y actuación (ver Figura 2.3) [25]. La interfaz electrónica recibe energía de un dispositivo de almacenamiento de energía (batería), el cual se conecta directamente al subsistema de alimentación. La energía de la batería se transmite a los otros subsistemas a través de los circuitos de protección, de acondicionamiento y reductores de voltaje que se encuentran en el subsistema de alimentación.

El subsistema de control contiene los elementos de computo y procesamiento necesarios para que el robot realice tareas de alto nivel. Primero, la computadora a bordo (OCU, por sus siglas en inglés) contiene el software y los algoritmos necesarios para mantener a los otros sistemas trabajando en conjunto y de manera ordenada. Por otro lado, el sistema embebido se utiliza para obtener

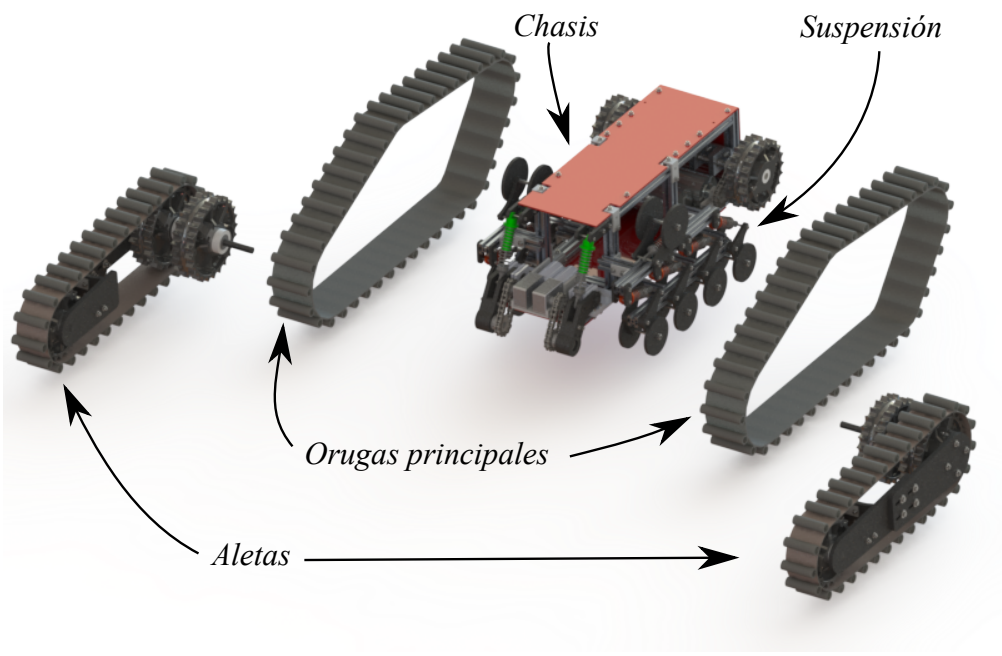


Figura 2.2. Subsistemas mecánicos principales del robot de búsqueda y rescate [24].

información de sensores a través de convertidores analógicos digitales (ADC, por sus siglas en inglés) o mediante puertos serial, I2C o SPI. Ambos componentes se comunican con el subsistema de sensado y actuación, permitiendo que el robot se mueva y perciba el entorno donde trabaja.

La OCU utiliza el sistema operativo de fuente abierta Ubuntu 14.04 LTS [26]. De igual manera, utiliza el Sistema Operativo Robótico (ROS, por sus siglas en inglés). ROS es un middleware flexible de fuente abierta que simplifica la tarea de crear comportamiento complejo y robusto en una amplia variedad de plataformas robóticas [27].

2.3. Visión por computadora

La visión por computadora es el análisis automático de imágenes y vídeos por computadoras a fin de que obtenga cierto grado de entendimiento del mundo [28]. El objetivo principal de la visión por computadora dentro de la robótica móvil es extraer información útil de imágenes que permita

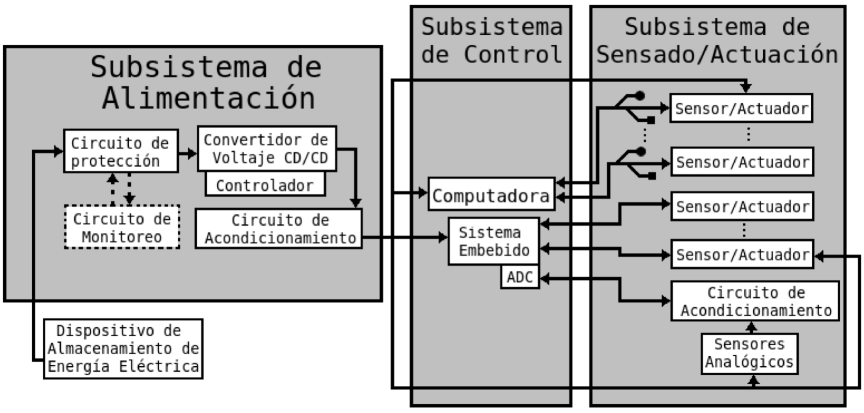


Figura 2.3. Diagrama a bloques de la interfaz electrónica [25].

a un robot realizar tareas de navegación. Esta es una tarea difícil debido a la complejidad de la representación de imágenes y el entorno en donde los robots se mueven.

Dentro de la robótica, la visión por computadora se utiliza principalmente para el reconocimiento de objetos o algún tipo de detección. Por ejemplo, reportar qué objeto se encuentra en una imagen, generar cuadros delimitadores alrededor de objetos o etiquetar cada pixel en imagen de acuerdo al objeto al que pertenece.

2.3.1. Fundamentos

En términos simples, una imagen puede ser definida como una función bidimensional de la luz y la intensidad, indicada por $f(x,y)$, donde x y y son coordenadas espaciales y la amplitud de f es la intensidad de la imagen en el punto (x,y) [29]. Cuando los valores de x, y, y f son cantidades finitas y discretas, entonces a esa imagen se le conoce como imagen digital.

En el caso de las computadoras, las imágenes se representan utilizando matrices numéricas que contienen la intensidad de los pixeles. A causa de esto, surge un problema dentro de la visión por computadora conocido como *brecha semántica*. Esta brecha es la diferencia entre cómo un humano percibe una imagen en comparación a cómo se representa de tal manera que una computadora entienda el proceso [29].

2.3.2. Detección de bordes y lineas

Los bordes en una imagen surgen en los limites entre regiones de diferente color, intensidad o textura. Un borde se puede ver como una ubicación de variación rápida de intensidad [30]. Los métodos más comunes para la detección de bordes incluyen el uso de el gradiente o laplaciana [29]. El operador gradiente G aplicado sobre una imagen se define como:

$$\nabla f(x,y) = [G_x\, G_y] = \left[\frac{\partial f}{\partial x}\ \frac{\partial f}{\partial y}\right] \tag{2.1}$$

y representa la variación máxima de intensidad en el punto (x,y). El modulo y dirección del gradiente están dados por:

$$|\nabla f(x,y)| = \sqrt{G_x^2 + G_y^2} \tag{2.2}$$

$$\angle\nabla f(x,y) = \arctan\left(\frac{G_x}{G_y}\right) \tag{2.3}$$

donde la dirección del gradiente es perpendicular al borde. Tomando en cuenta el coste computacional de la ecuación 2.2, esta se simplifica utilizando:

$$|\nabla f(x,y)| = G_x^2 + G_y^2 \tag{2.4}$$

Para el calculo del gradiente, comúnmente se utilizan operadores (mascaras) que son poco sensibles al ruido tales como los operadores de Roberts, Prewitt y de Sobel.

En comparación con el gradiente, la Laplaciana utiliza la segunda derivada, por lo tanto, este operador es muy sensible al ruido y se utiliza con menos frecuencia [31]:

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial^2 x} + \frac{\partial f^2}{\partial^2 y} \tag{2.5}$$

Además de estos operadores, existen otras técnicas más populares para la detección de bordes,

una de ellas siendo el detector de Canny. Este detector fue el primero en demostrar que el filtrado Gaussiano es óptimo para la detección de bordes ya que ayuda a reducir ruido y evitar falsos positivos. Por otra parte, para mejorar la localización de los bordes, este detector realiza un proceso llamado supresión de no máximos, el cual adelgaza los bordes generados por el gradiente.

El detector de bordes Marr-Hildreth también es muy utilizado y, al igual que el de Canny, utiliza filtrado Gaussiano para reducir el ruido. Sin embargo, en comparación con el detector de Canny, utiliza el Laplaciano para posteriormente realizar una búsqueda de los puntos de cruce por cero y obtener los bordes [32].

Los detectores mencionados hasta ahora solamente toman en cuenta la información local del entorno de cada pixel. En comparación, la transformada de Hough (HT, por sus siglas en inglés) es una técnica desarrollada para la extracción de lineas, círculos y elipses [32] y se basa en la información que suministra toda la imagen [31].

La HT para la extracción de rectas utiliza la ecuación de una linea recta dada por:

$$x\cos(\theta) + y\sin(\theta) = \rho \tag{2.6}$$

la cual define a una recta por sus parámetros θ y ρ. Esto significa que todos los puntos que pertenezcan a la misma recta tendrán las mismas componentes θ y ρ. Si se grafican todos los valores posibles de θ y ρ definidos por cada punto con coordenadas (x,y) en el plano cartesiano, entonces se obtiene lo que se conoce como el espacio de Hough. El resultado de la HT son funciones sinusoidales que corresponden con puntos en el espacio cartesiano de la imagen. A causa de esto, los puntos que son colineales (en el espacio cartesiano de la imagen) tendrán curvas que se intersectan en un punto (ρ,θ) en el espacio de Hough.

Para detectar las lineas rectas, la HT utiliza un arreglo conocido como acumulador. El número de parámetros determina la dimensión del acumulador, que en este caso (ecuación 2.6) son dos. Un procedimiento muy común que se utiliza para obtener los valores del acumulador inicia con una imagen con los bordes previamente detectados y posteriormente utiliza los puntos (x,y) de la

imagen para obtener el valor de ρ para cada valor de θ en un rango de cero a 180 grados. Cada vez que un punto coincida con las coordenadas (ρ, θ), entonces se incrementa el valor del acumulador en esa ubicación. Los máximos locales del acumulador representan una fuerte evidencia de que existen lineas rectas en esas ubicaciones [32].

La ventaja principal de la HT es que utiliza toda la información de la imagen y, por lo tanto, se vuelve menos inmune a perdidas parciales de los bordes. Sin embargo, este método tiene un coste computacional grande y las rectas que detecta son infinitas.

2.4. Aprendizaje automático

El aprendizaje automático se puede definir como la habilidad de una máquina de adquirir conocimiento extrayendo patrones de datos crudos sin necesidad de ser programadas explícitamente. El desempeño de estos algoritmos dependen en gran medida de la representación de los datos que se les proporciona y muchos problemas de inteligencia artificial se resuelven eligiendo la representación más adecuada de los datos[33]. Sin embargo, la elección del tipo de representación es compleja y los algoritmos para aprendizaje de representaciones se vuelven igual de complejos que el problema inicial. Para resolver este problema, el aprendizaje profundo introduce representaciones que se expresan en términos de otras representaciones más simples.

Aprendizaje supervisado

Durante el aprendizaje supervisado, los datos que se utilizan para el algoritmo incluyen las soluciones deseadas, mejor conocidas como etiquetas (Figura 2.4). La tarea más común para los algoritmos de aprendizaje automático es la clasificación, en donde el algoritmo debe especificar a que categoría pertenece una entrada. En el caso de la Figura 2.4, el algoritmo tendría que especificar si un correo electrónico es deseado o no. Estos algoritmos también se utilizan para predecir un valor numérico a partir de características. Por ejemplo podrían utilizar el kilometraje, el año y la marca de un vehículo para predecir el precio del mismo. A este tipo de tareas se le llama regresión y

para entrenar al sistema se necesitaría las características del vehículo y sus precios (etiquetas). Los algoritmos más importantes para el aprendizaje supervisado incluyen [34]:

- K-vecinos más próximos
- Regresión lineal
- Regresión logística
- Máquinas de vectores de soporte
- Árboles de decisiones
- Redes neuronales

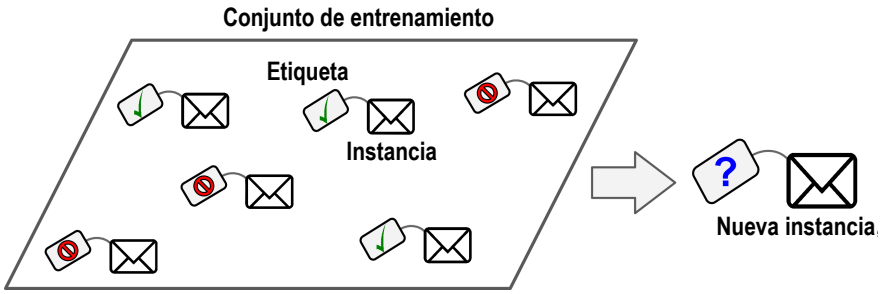


Figura 2.4. Aprendizaje supervisado [34].

Aprendizaje no supervisado

Durante el aprendizaje no supervisado, los datos que se utilizan para entrenamiento no tienen etiquetas, por lo tanto el sistema intenta aprender sin etiquetas. Los algoritmos más importantes para el aprendizaje no supervisado incluyen [34]:

- Clustering
- Detección de anomalías y novedades
- Visualización y reducción de dimensionalidad
- Aprendizaje de asociación de reglas

Durante la ejecución de los algoritmos de clustering, se detectan grupos de datos similares sin ayuda externa como etiquetas. Por ejemplo, utilizando una base de datos sobre visitantes en un blog, se puede utilizar las características (*features*) de los visitantes para implementar un algoritmo de clustering y agrupar visitantes similares (Figura 2.5).

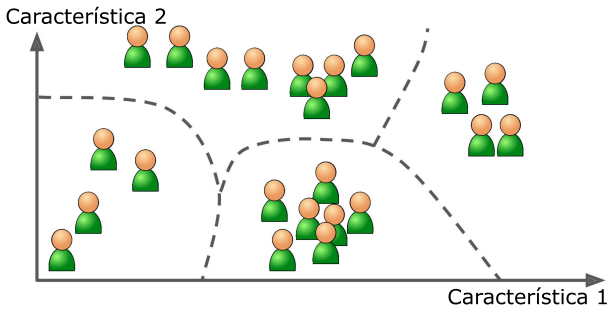


Figura 2.5. Ejemplo de clustering [34].

Por otra parte, a los algoritmos de visualización, se les provee información compleja y sin etiquetas y se obtiene una representación gráfica en 2D o 3D que permite ser graficada (Figura 2.6). Los algoritmos de este tipo tratan de mantener los grupos separados en la visualización y, de tal manera, permitir una mejor comprensión de cómo los datos están organizados. Algunas veces se implementan algoritmos de reducción de dimensión, los cuales tienen como objetivo simplificar los datos sin perder mucha información. Por ejemplo, el kilometraje de un automóvil está muy relacionado con su antigüedad, por lo tanto, durante la reducción de dimensión se unirán estas características en una. A esta tarea se le conoce como extracción de características.

La detección de anomalías se utiliza para encontrar valores atípicos en una base de datos antes de utilizarlos en otro algoritmo de aprendizaje. Durante el entrenamiento de este sistema, solamente se le muestran datos normales de tal manera que cuando se le muestre un dato nuevo, se detectará o no como anomalía (ver Figura 2.7). La detección de novedades es muy similar, sin embargo este algoritmo es menos tolerante y necesita ser entrenado solamente con datos normales.

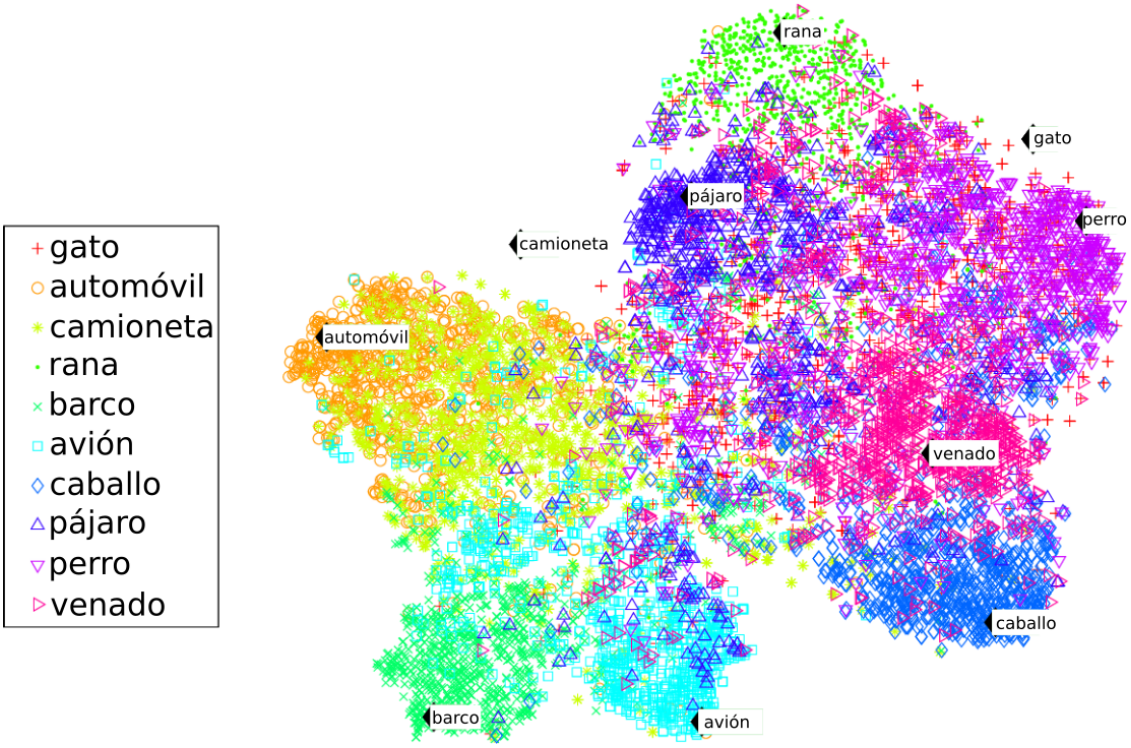


Figura 2.6. Ejemplo de visualización [34].

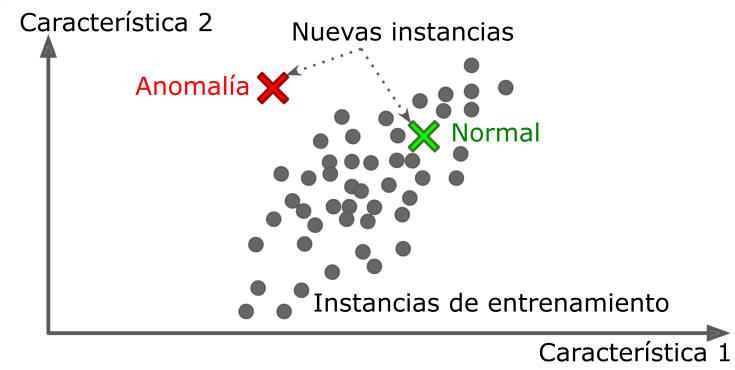


Figura 2.7. Ejemplo de detección de anomalías [34].

Tareas comunes del aprendizaje automático

El aprendizaje permite que se realicen tareas que son muy complejas de solucionar utilizando programas escritos e implementados por humanos. Algunas de las tareas que pueden resolverse con algoritmos de aprendizaje automático se presentan a continuación.

Clasificación

En este tipo de tarea, el algoritmo tiene que decidir a cual de las k categorías pertenece una

entrada. Para realizar esta tarea, el algoritmo tiene que producir una función $f : \mathbb{R}^n \to \{1, \ldots, k\}$. Cuando se tiene $y = f(x)$, el modelo le asigna una categoría y a una entrada dada por un vector de características $x \in \mathbb{R}^n$. El valor de y puede ser un valor numérico que representa una categoría o puede ser una probabilidad de pertenecer a una clase o categoría. Un ejemplo de una tarea de clasificación es el reconocimiento de objetos, en donde la entrada es una imagen y la salida es un código numérico, el cual identifica al objeto. Este tipo de tareas tienden a tener mejores resultados al utilizar aprendizaje profundo.

Regresión

Esta tarea es similar a clasificación, sin embargo ahora se le pide al programa predecir un valor numérico dependiendo de una entrada. Para resolver esta tarea, el algoritmo de aprendizaje obtiene una función $f : \mathbb{R}^n \to \mathbb{R}$. Un ejemplo de regresión es determinar el valor monetario de un vehículo considerando sus características tales como su edad y kilometraje.

Transcripción

Durante el sistema de aprendizaje automático tiene como entrada datos sin estructura y tiene que transcribir la información en una forma textual discreta. Un ejemplo es cuando al sistema se le proporciona una imagen que contiene texto y de esta manera el algoritmo regresa el texto en formato ASCII.

2.5. Redes neuronales artificiales

Las redes neuronales artificiales son inspiradas por la arquitectura del cerebro humano. Estos sistemas son versátiles, poderosos y escalables, lo que permite que realicen problemas complejos como clasificación de imágenes o reconocimiento de voz.

A las redes neuronales más simples se les conoce como perceptrones. Este tipo de red neuronal contiene una capa simple de entradas $x_1 \ldots x_5$ y un nodo de salida o *Threshold Logic Unit* (TLU, por sus siglas en inglés) y puede o no tener una neurona de bias b (Figura 2.8). La capa simple transmite directamente las entradas al nodo de salida, el cual multiplica los pesos $w_1 \ldots w_5$ de cada

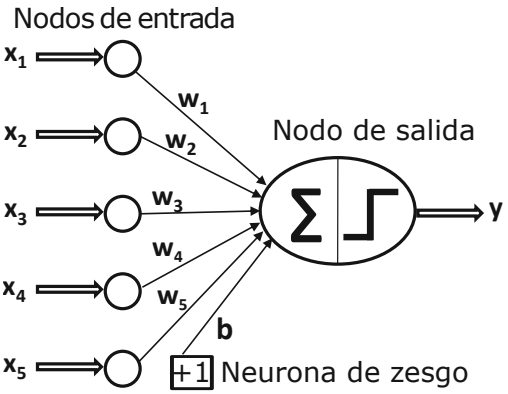


Figura 2.8. Perceptrón con zesgo [35].

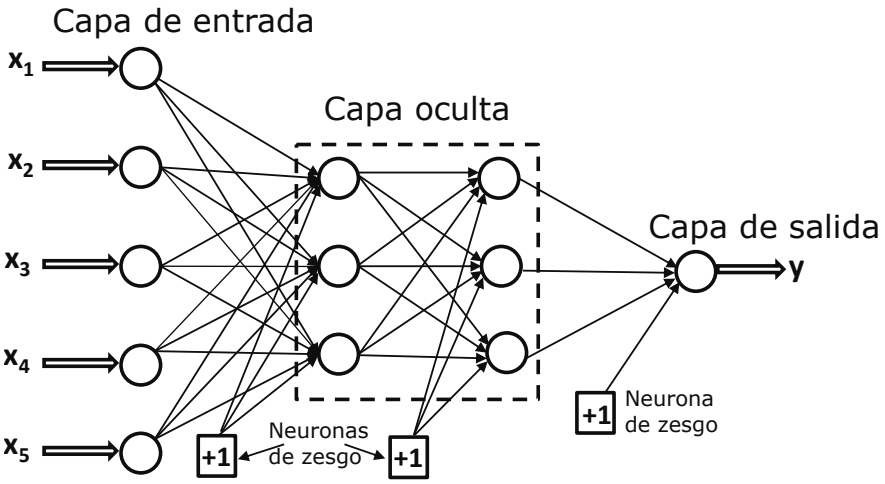


Figura 2.9. Perceptrón multicapa con zesgo [35].

entrada y suma cada resultado para determinar si el nodo se activará [35].

Al agregar más capas de TLUs (capas ocultas), se obtiene un perceptrón multicapa. Su arquitectura básica se muestra en la Figura 2.9, la cual incluye una capa de entrada, una oculta y la de salida. Las redes neuronales con esta arquitectura son mejor conocidas como redes neuronales *feedforward*.

2.5.1. Redes neuronales feedforward

Las redes neuronales *feedforward*, también conocidas como redes profundas *feedforward* o perceptrones multicapa (MLPs, por sus siglas en inglés) son los modelos más utilizados en aplicaciones comerciales. Estos modelos se llaman así debido a que carecen de algún mecanismo de

retroalimentación [36]. En otras palabras, no existen conexiones que permitan que las salidas del modelo se retroalimenten al mismo. Cuando una red neuronal incluye esta retroalimentación, entonces se le conoce como red neuronal recurrente (RNN, por sus siglas en inglés).

Una red neuronal *feedforward* recibe el nombre de red ya que se representa por una composición de múltiples funciones. Por ejemplo, una función compuesta de tres funciones: $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}))$, donde a $f^{(1)}$ se le conoce como la primer capa, a $f^{(2)}$ la segunda capa, etcétera. El tamaño de la composición determina la *profundidad* del modelo y es así como surgió el término de aprendizaje profundo o *deep learning*.

Durante el entrenamiento de una red neuronal, se busca aproximar la función $f(\mathbf{x})$ a la función $f^*(\mathbf{x})$. Para hacer esto, se utilizan ejemplos de entrenamiento $\mathbf{x}$ acompañados por una etiqueta $y \approx f^*(\mathbf{x})$. Con estos ejemplos se busca que, para cada valor de $\mathbf{x}$, se obtenga un valor muy cercano a y. Los datos de entrenamiento especifican, de manera directa, el comportamiento deseado del modelo, sin embargo, no especifican las salidas deseadas de cada capa, por lo tanto, a estas capas se les conoce como capas ocultas.

2.5.2. Redes neuronales convolucionales

Las redes neuronales convolucionales (CNNs, por sus siglas en inglés) se utilizan para procesar datos que tienen una estructura cuadriculada, por ejemplo, series de tiempo tomadas con un mismo intervalo de muestreo (cuadricula de una dimensión) o imágenes (cuadricula de dos dimensiones).

Este tipo de redes utilizan una operación matemática lineal llamada convolución, la cual se puede definir utilizando un ejemplo en donde se tiene una función de distancia $x(t)$ la cual está definida en cualquier instante de tiempo t. Las mediciones tienen cierto ruido, por lo tanto, se busca disminuirlo realizando un promedio ponderado de varias mediciones. Para realizar esto se utiliza una función $w(a)$, donde a es la edad del dato. Si se aplica esta operación de promedio ponderado, entonces se obtiene una nueva función s que provee una estimación de la posición

[33]:

$$s(t) = \int x(a)w(t-a)da \tag{2.7}$$

Esta operación se conoce como convolución y se denota comúnmente con un asterisco:

$$s(t) = (x * w)(t) \tag{2.8}$$

En este ejemplo se debe considerar que w es una función de densidad de probabilidad, de lo contrario la salida no será un promedio ponderado. También, w tiene que ser 0 para todos los valores negativos (cuando $t < a$), de lo contrario se estaría intentado obtener valores futuros, lo cual es imposible. En el ejemplo, a $x(t)$ se le conoce como la entrada, a w como el kernel y a la salida comúnmente se le conoce como mapa de características.

Cuando se trabaja con datos en una computadora, el tiempo y las mediciones se discretizan, por lo tanto se puede definir la convolución discreta como:

$$s(t) = (w * x)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.9}$$

En aplicaciones de aprendizaje automático, la entrada es un arreglo multidimensional de datos, y el kernel es un arreglo multidimensional de parámetros que es adaptado por el algoritmo de aprendizaje [33]. Usualmente, a estos arreglos se les conoce como tensores y, debido a que cada elemento de la entrada y del kernel se tienen que guardar por separado, usualmente se asume que las funciones son cero excepto por los valores finitos guardados. Por lo general se implementa la suma infinita de la ecuación 2.9 como una suma finita de los valores.

La convolución se puede utilizar en más de dos ejes a la vez, por ejemplo, si se utiliza una imagen I de dos dimensiones como entrada, entonces probablemente se utilizaría un kernel K de dos dimensiones:

$$S(i,j) = (I * K)(i,j) = \sum_{M}\sum_{N} I(m,n)K(i-m, j-n) \tag{2.10}$$

La convolución es conmutativa, por lo tanto la ecuación 2.10 se puede escribir como:

$$S(i,j) = (K * I)(i,j) = \sum_M \sum_N I(i-m, j-n)K(m,n) \tag{2.11}$$

Esta propiedad de conmutatividad existe debido a que el kernel se voltea relativo a la entrada. Esta propiedad generalmente no es de gran importancia durante la implementación de redes neuronales. Por lo tanto, muchas librerías de redes neuronales implementan una función llamada correlación cruzada, la cual es igual que la convolución, pero sin voltear el kernel:

$$S(i,j) = (K * I)(i,j) = \sum_M \sum_N I(i+m, j+n)K(m,n) \tag{2.12}$$

En procesamiento de imágenes se utiliza la convolución para obtener efectos visibles y un conjunto de filtros convolucionales se combinan para formar una capa convolucional de una red neuronal [37].

2.5.3. Capas de CNNs

Existen distintos tipos de capas que se pueden utilizar para diseñar una CNN, sin embargo, las más utilizadas son las capas convolucionales, completamente conectadas, de normalizacion por lotes (BN, por sus siglas en inglés) y de *dropout*.

Las capas convolucionales de una CNN generalmente consisten de tres etapas (ver Figura 2.10). En la primer etapa, se realizan convoluciones en paralelo para producir un conjunto de activaciones lineales. En la segunda etapa, cada activación se ejecuta a través de una función de activación no lineal, cómo por ejemplo, una función de activación lineal rectificada (ReLU, por sus siglas en inglés). Esta etapa a veces es conocida como la etapa de detección. Finalmente, en la última etapa se realiza un submuestreo (*pooling*), lo que ayuda a disminuir la varianza causada por pequeñas traslaciones en la imagen de entrada [33]. La representación de la Figura 2.10 considera tres etapas en una capa convolucional, sin embargo, existen otras representaciones en donde cada etapa de

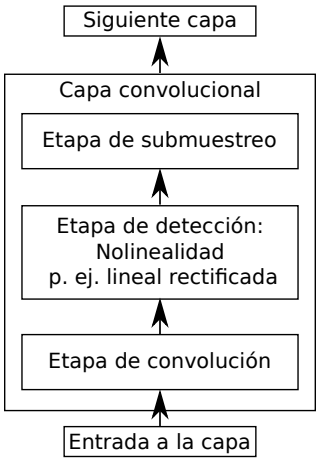


Figura 2.10. Estructura básica de una capa convolucional [33].

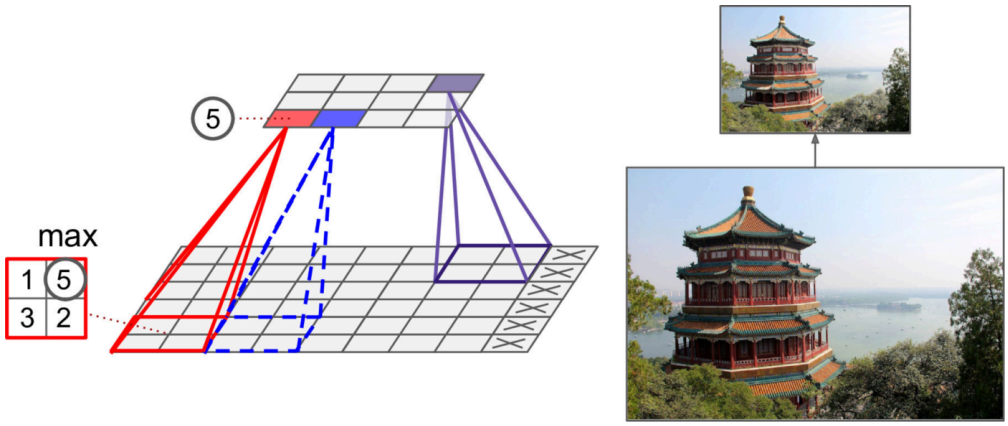


Figura 2.11. Etapa de submuestreo tipo *max pooling* [34].

procesamiento recibe el nombre de capa.

La etapa de submuestreo también sirve para reducir la carga de computo, el uso de memoria y el número de parámetros de la red neuronal. En la Figura 2.11 se muestra la operación de *max pooling*, en donde se utiliza un kernel de submuestreo (sin pesos), un paso (cuantos pixeles se traslada) de 2 y sin *padding* (rellenado de ceros alrededor de la imagen). Como se mencionó anteriormente, al utilizar etapas de submuestreo, en las capas de CNNs, es posible introducir un nivel de invarianza a pequeñas traslaciones. Sin embargo, existen algunas desventajas al utilizar esta operación. Primero, se pierde información al disminuir el tamaño de la entrada. Además, en ciertas aplicaciones, la invarianza no es deseada, por ejemplo, para la segmentación semántica donde se clasifica cada pixel dependiendo del objeto al que pertenece [34].

Durante el entrenamiento de una red neuronal profunda surge un fenómeno conocido como *internal covariance shift* y sucede cuando la distribución de las entradas de cada capa cambia a causa del cambio de los parámetros de la capa anterior. Este fenómeno causa que el entrenamiento sea lento y que la inicialización de los parámetros tenga que hacerse de manera cuidadosa. Para disminuir estos efectos, se introdujo una técnica conocida como normalización por lotes o *batch normalization* (BN por sus siglas en inglés) que se encarga de nornalizar dichas entradas para cada mini-lote de entrenamiento [38]. Las entradas normalizadas $\hat{x}$ de las entradas x se obtienen utilizando la ecuación 2.13.

$$\hat{x_i} = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \varepsilon}} \tag{2.13}$$

Donde, utilizando cada mini-lote β, los valores de μ_β y σ_β se obtienen durante el entrenamiento:

$$\mu_\beta = \frac{1}{M}\sum_{i=1}^{m} x_i \tag{2.14}$$

$$\sigma_\beta = \frac{1}{m}\sum_{i=1}^{m} \left(x_i - \mu_\beta\right)^2 \tag{2.15}$$

El valor de ε en la ecuación 2.13 es igual a un valor muy pequeño, por ejemplo $1e-7$ para evitar el cálculo de la raíz cuadrada de cero. En una CNN, una capa de *batch normalization* genera salidas que tienen aproximadamente una media igual a cero y una desviación estándar de uno. Al utilizar una capa de BN se pueden utilizar tasas de aprendizaje más grandes y, algunas veces, elimina la necesidad de utilizar capas *dropout*.

Finalmente, las capas completamente conectadas son las que se están conectadas a todas las neuronas de la capa previa. Generalmente, en una CNN, estas capas se utilizan al final de la red. Es muy común utilizar múltiples capas completamente conectadas antes de utilizar una capa final con un clasificador *softmax* [36].

2.5.4. Entrenamiento

Para entrenar una red neuronal profunda se utiliza un algoritmo conocido como *backpropagation* o propagación hacia atrás. Este algoritmo es de gran importancia debido a que permite calcular el gradiente de la función de costo con respecto a los diferentes parámetros de la red y al combinar este algoritmo con una técnica de optimización (por ejemplo Gradiente Descendente) se obtiene el algoritmo de entrenamiento que permite determinar de qué manera se deben ajustar los parámetros para disminuir el error [34].

El primer paso del entrenamiento consiste en pasar todos los datos de entrenamiento de un lote a través de la red neuronal con parámetros previamente inicializados. Durante esta etapa o paso hacia delante (en inglés *forward pass*) todos los resultados intermedios de las capas son almacenados en memoria ya que son necesarios para realizar el paso hacia atrás.

La salida de la red neuronal es una predicción de la etiqueta del dato de entrenamiento. Utilizando esta información, el siguiente paso calcula el valor de la función de costo o bien el error que ha cometido el modelo durante la clasificación.

Después de calcular el valor de la función de costo, con el algoritmo de *backpropagation*, se calcula el gradiente respecto a todos los parámetros de la red y se determina la contribución de cada neurona al error total del modelo. Finalmente, utilizando el gradiente obtenido con *backpropagation*, se utiliza alguna técnica de optimización para realizar el ajuste de los parámetros del modelo y minimizar el error de la red [36].

2.5.5. Arquitecturas de CNNs

Las CNNs generalmente utilizan múltiples capas convolucionales como las que se muestran en la Figura 2.10. La imagen original se reduce en tamaño cada vez que pasa por una capa convolucional, pero también se vuelve más profunda y, por lo tanto, se extraen más características. Al final de la red se tiene una red neuronal *feedforward*, compuesta de capas completamente conectadas, y una capa que devuelve la predicción, por ejemplo, una capa *softmax* que tiene como salida probabilidad

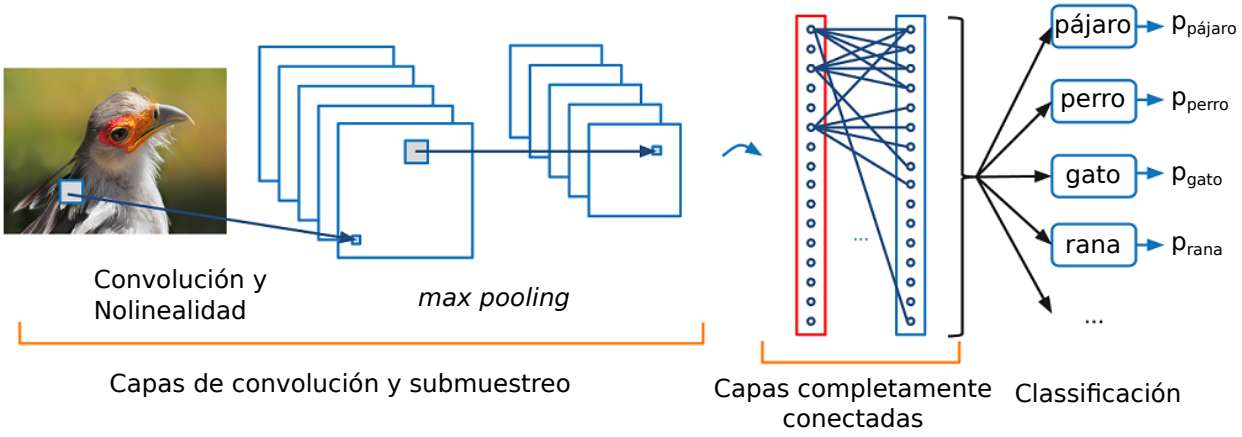


Figura 2.12. Arquitectura básica de una CNN [37].

de estar en cierta clase. En la Figura 2.12 se muestra la arquitectura básica de una red neuronal convolucional.

Durante el paso de los años, se han desarrollado distintos modelos basados en la arquitectura simple de una CNN. Con el desarrollo de nuevos se han generado grandes avances en el área de la inteligencia artificial. A continuación se presentan las arquitecturas más utilizadas.

LeNet-5

La arquitectura LeNet-5 fue creada por Yann LeCun en 1998 [39]. Esta es una de las arquitecturas más conocidas y ha sido ampliamente utilizada para el reconocimiento de dígitos escritos a mano. La red está compuesta de las capas que se muestran en la Figura 2.13. La entrada es una imagen con un tamaño de 28x28 pixeles, pero es rellenada con ceros para obtener una imagen de 32x32. Además, para las etapas de muestreo, utiliza capas de pooling promedio (*average pooling*), las cuales calculan el promedio de la entrada, la multiplican por un coeficiente (obtenido durante el aprendizaje), le suman un término de zesgo y finalmente se ejecuta a través de una función de activación.

En la Figura 2.13 se muestran los mapas de características C3, C5 y C7, los cuales, son los resultados de cada capa de convolución. Por otra parte, los mapas de características S2 y S4 son los resultados de las capas *average pooling*. Finalmente, en la capa de salida, cada neurona calcula

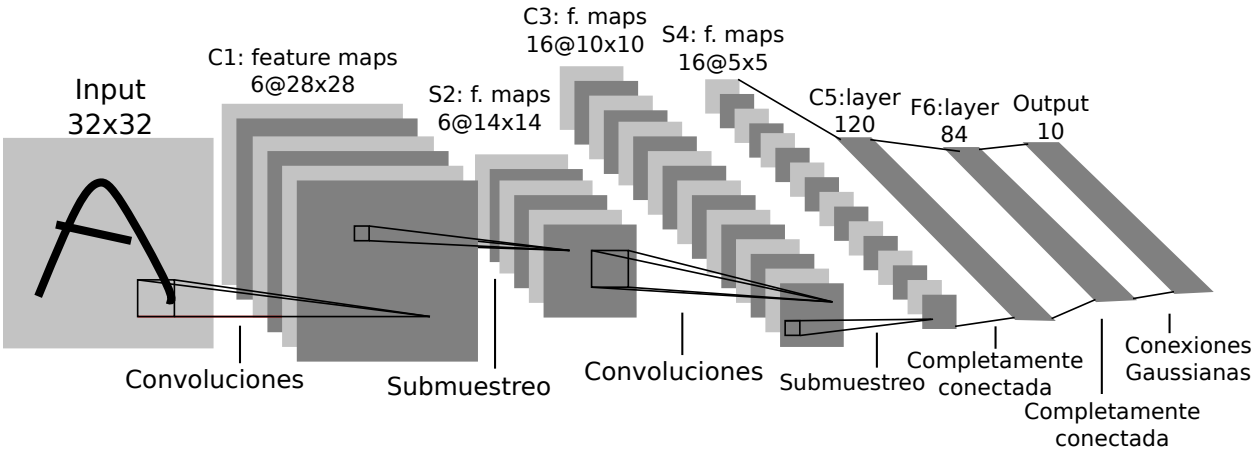


Figura 2.13. Arquitectura de la red neuronal LeNet-5 [39].

la distancia Euclidiana entre el vector de entrada y sus pesos. Cada salida determina el nivel de pertenencia, del dígito de entrada, a cierta clase [39].

AlexNet

La arquitectura AlexNet fue desarrollada por Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton para clasificar 1.2 millones de imágenes en la competencia de ImageNet LSVRC-2010 [40]. Esta red neuronal tiene 60 millones de parámetros y 650,000 neuronas y está conformada de cinco capas convolucionales, algunas seguidas de capas *max-pooling*, tres capas completamente conectadas y finalmente una capa *softmax* (ver Figura 2.14). Esta red es similar a la LeNet-5, sin embargo, es más grande y más profunda. Además, la AlexNet fue una de las primeras arquitecturas que utilizó capas convolucionales consecutivas, en lugar de alternar capas convolutivas y de submuestreo (*pooling*) [34].

Para reducir el sobreajuste, esta arquitectura utiliza dos métodos de regularización. Primero, aumenta la cantidad de imágenes de entrenamiento al aplicar transformaciones que no cambian la etiqueta, como traslaciones, reflexiones o cambios en la intensidad de los canales RGB. Segundo, utiliza el método de regularización conocido como *dropout*, el cual consiste en hacer cero la salida de ciertas neuronas y de tal manera evitar el sobreajuste. La probabilidad de que a una neurona sea abandonada (*dropped out*) es un parámetro de la red que se define antes del entrenamiento [40].

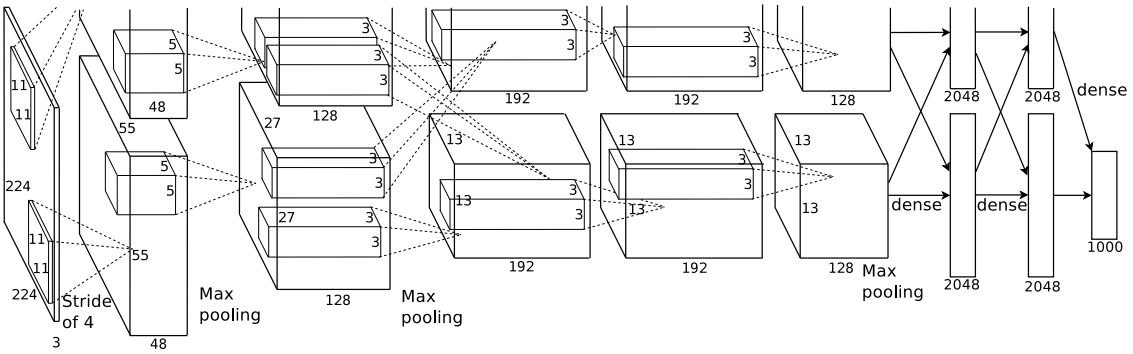


Figura 2.14. Arquitectura de la red neuronal AlexNet [40].

GoogLeNet

Esta arquitectura fue desarrollada por Christian Szegedy et al. de Google Research para el *Large Scale Visual Recognition Challenge* (ILSVRC) del 2014. GoogLeNet es una de las arquitecturas más profundas debido al número total de capas, 22 contando solamente las que tienen parámetros. Para lograr esta profundidad, utilizan módulos denominados *Inception*, en donde utilizan convoluciones con kernels de tamaño 1×1. En comparación con la arquitectura LeNet-5, GoogLeNet utiliza 12 veces menos parámetros (aproximadamente 4 millones) y al mismo tiempo es más precisa [41].

VGGNet

La arquitectura VGGNet fue desarrollada por Karen Simonyan y Andrew Zisserman para el ImageNet Challenge del 2014 [42]. Esta arquitectura es simple y utiliza una secuencia de capas que alterna dos o tres capas de convolución con una capa de *pooling*. La red tiene un total de 16 capas convolucionales, las cuales unicamente utilizan *kernels* de 3x3. Al final de la red utiliza dos capas completamente conectadas y un clasificador *softmax*.

2.5.6. Aprendizaje por transferencia

En general, no es recomendable entrenar redes neuronales profundas (DNNs, por sus siglas en inglés) de gran tamaño desde cero, en lugar, se debería buscar una red que realice una tarea similar

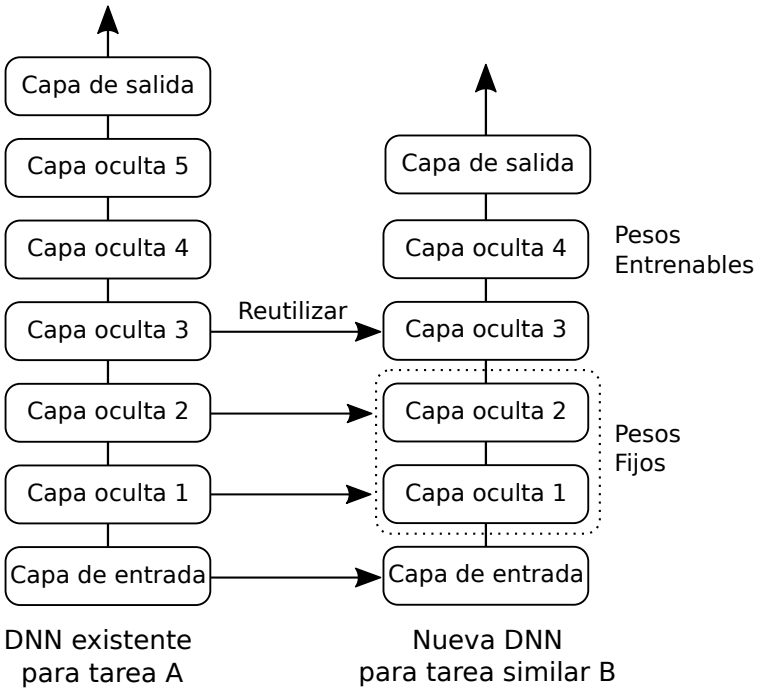


Figura 2.15. Ejemplo de *Transfer learning* al reutilizar capas de una DNN [34].

a la que se pretende realizar y utilizar parte de esta red (usualmente la parte inicial). A esta técnica se le conoce como aprendizaje por transferencia (*transfer learning*). Con *transfer learning* no solamente se disminuye el tiempo de entrenamiento, sino también la cantidad de datos necesarios para el mismo [34].

En la Figura 2.15 se muestra un ejemplo simple de *transfer learning*. En la figura se muestran dos DNNs, una diseñada para una tarea A y otra para una tarea B. La red para la tarea A consiste de una capa de entrada, cinco capas ocultas y una capa de salida. La tarea B utiliza una red neuronal que realiza una tarea similar, por lo tanto, se reutilizan las primeras cuatro capas de la red neuronal hecha para la tarea B. En general, solamente se reutilizan las capas bajas de una red debido a que extraen características más generales. Por otra parte, las capas altas tienden a extraer características especificas, y además puede que la capa de salida no tenga la forma correcta (número de salidas) para la nueva tarea. Otra cosa que se debe tomar en cuenta es que se pueden definir capas que utilizan pesos fijos y, por lo tanto, no cambiaran durante el entrenamiento de la nueva DNN [34].

2.6. Detectores de objetos

La tarea de clasificar y localizar objetos en una imagen se conoce como detección de objetos. Dentro del aprendizaje automático, un algoritmo de clasificación se encarga de determinar a cual de las *k* categorías pertenece una entrada. Algunos de las arquitecturas más utilizadas para la detección de objetos incluyen YOLO (*You Only Look Once*), SSD (*Single Shot Detector*) y Faster R-CNN (*Region based Convolutional Neural Networks*) [34].

2.6.1. Arquitectura de un detector de objetos

La arquitectura básica de un detector de objetos, basado en aprendizaje profundo, se muestra en la Figura 2.16 y consiste de dos partes principales, la espina vertebral (*backbone*) y la cabeza (*head*) [43]. El *backbone*, normalmente, está conformado de una CNN y se encarga de generar mapas de características a partir de la imagen de entrada. Los detectores que utilizan una Unidad de Procesamiento Grafico (GPU por sus siglas en inglés) pueden utilizar *backbones* tales como VGG, ResNet, ResNeXt o DenseNet. Por otra parte, los que utilizan solamente una Unidad Central de Procesamiento (CPU por sus siglas en inglés) pueden utilizar *backbones* tales como SqueezeNet, MobileNet o ShuffleNet.

La cabeza de un detector de objetos se encarga de realizar las predicciones de los cuadros delimitadores y sus clases correspondientes. La cabeza de un detector determina si el detector de objetos es de una etapa (*one-stage object detector*) o de dos etapas (*two-stage object detector*). Los detectores de objetos de una etapa realizan las predicciones utilizando solamente capas convolucionales. En comparación, los de dos etapas primero utilizan un algoritmo o red neuronal para generar propuestas de regiones en donde pueden existir objetos y posteriormente, utilizando una red neuronal para clasificación, determinan la clase del objeto (si es que existe) dentro del cuadro delimitador. Debido a esta diferencia, normalmente, los detectores de una etapa tienden a ser más rápidos.

Los detectores de objetos actuales tienden a incluir algunas capas adicionales entre la cabeza y

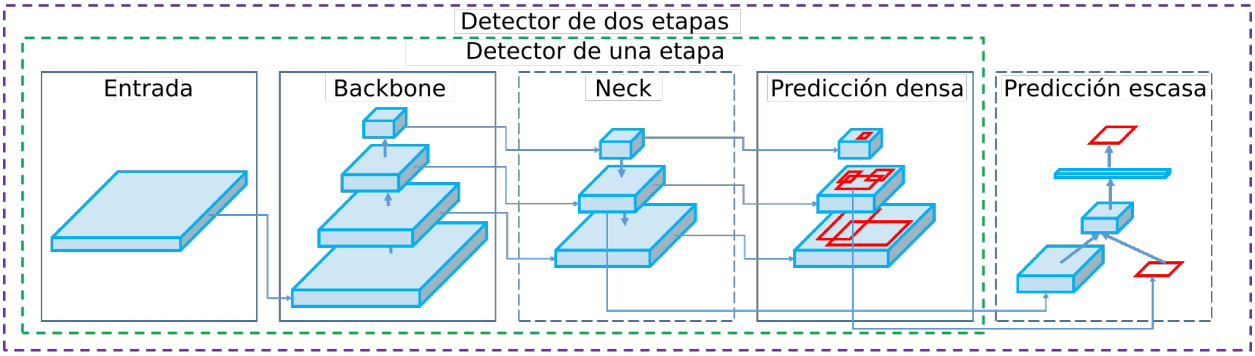


Figura 2.16. Arquitectura de un detector de objetos [43].

el *backbone*. A este conjunto de capas se le conoce como cuello (*neck*) y normalmente se encargan de unir mapas de características de diferentes etapas del *backbone*. Las redes neuronales que incorporan este mecanismo incluyen Feature Pyramid Network (FPN), Path Aggregation Network (PAN), BiFPN y NAS-FPN.

A continuación se describen las arquitecturas de los detectores de objetos utilizados en este trabajo.

2.6.2. Faster R-CNN

El detector de objetos Faster R-CNN [44] proviene de la familia de R-CNN que ha pasado por múltiples iteraciones. Faster R-CNN es el detector más actual y su arquitectura se muestra en la Figura 2.17. Esta arquitectura tiene a ResNet50 [45] como *backbone*, la cual contiene 48 capas convolucionales, una capa de *max pooling* y una capa completamente conectada. ResNet introduce el concepto de aprendizaje residual, el cual busca reducir el problema de degradación de la exactitud (*accuracy*) al utilizar conexiones tipo corto circuito (*shortcut conections* en Inglés). Las conexiones de corto circuito son aquellas que saltan una o más capas dentro de la red, como se muestra en la Figura 2.18. En el caso de ResNet, la conexión no realiza ninguna transformación sobre la entrada y, de esta manera, no se incrementan el número de parámetros o la complexidad computacional. Al hacer uso de esta conexión, se pueden generar redes neuronales más profundas y más fáciles de optimizar al disminuir el problema de degradación mencionado anteriormente.

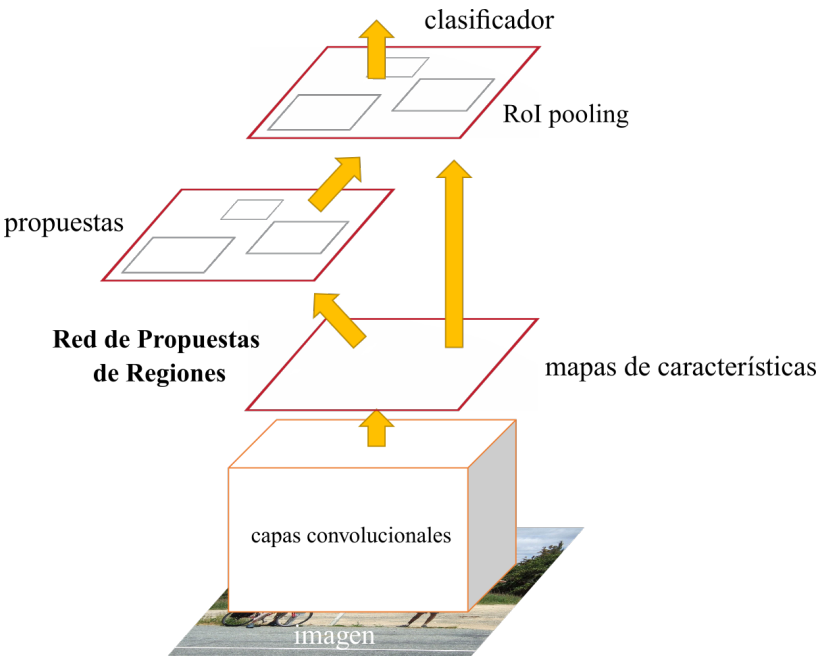


Figura 2.17. Arquitectura del detector Faster-RCNN ResNet50 [44].

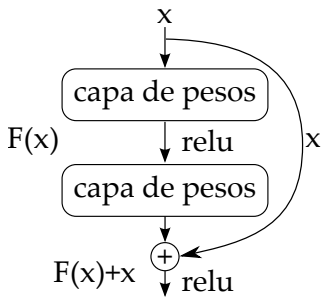


Figura 2.18. Bloque utilizado para el aprendizaje residual [45].

Posteriormente, Faster R-CNN utiliza los mapas de características generados por el *backbone* para realizar una predicción densa de la ubicación de los objetos. Esta predicción densa se realiza utilizando Red de Propuestas de Regiones (RPN, por sus siglas en ingés). La RPN toma el mapa de características (generado por el *backbone*) y genera un conjunto de regiones rectangulares en donde pueden existir objetos y sus respectivas certidumbres. Para realizar las propuestas, la red utiliza un sistema de ventanas deslizantes de tamaño *nxn* para obtener un mapa de características de dimensión reducida. Posteriormente, las características se utilizan en dos capas diferentes, una de regresión y una de clasificación. A partir de un número máximo de propuestas k, la capa de regresión utiliza el mapa de características de dimensión reducida y las cajas de anclaje para pro-

ducir $4k$ salidas, indicando las coordenadas de los k cuadros. Por otra parte, la capa de clasificación produce 2 probabilidades por cada caja de anclaje (en total $2k$), las cuales estiman la probabilidad de que exista o no un objeto en la propuesta. Las cajas de anclaje que se utilizan están centradas sobre cada ventana y estan asociadas con una escala y un relación de aspecto.

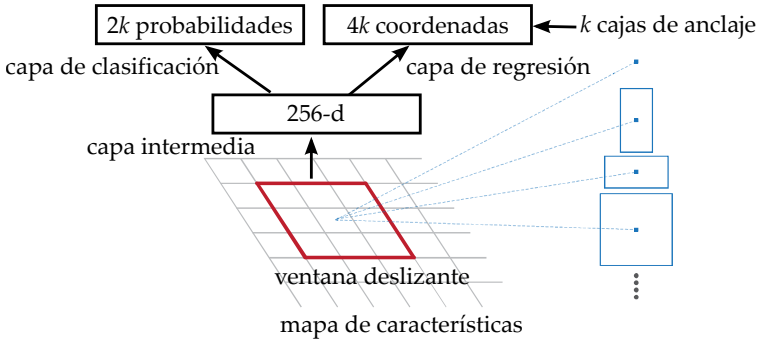


Figura 2.19. Red de Propuestas de Regiones (RPN) [44].

Finalmente, Faster R-CNN utiliza la arquitectura Fast R-CNN [46] para realizar las predicciones a partir del mapa de características generado utilizando ResNet50 y las regiones propuestas (también conocidas como ROIs) generadas utilizando RPN. Cada región (correspondiente a el mapa de características), se pasa por una capa de *pooling* y se genera un vector característico de longitud fija. Posteriormente, cada vector se pasa por múltiples capas completamente conectadas (FCs), y la salida resultante se utiliza en dos capas de salida distintas: una para capa tipo softmax para determinar la probabilidad de pertenecer a una de las $K+1$ clases (considerando el fondo como una clase) y otra que corrige las coordenadas de las cajas de anclaje para cada una de las K clases. En la Figura 2.20 se muestran los componentes principales de Fast R-CNN.

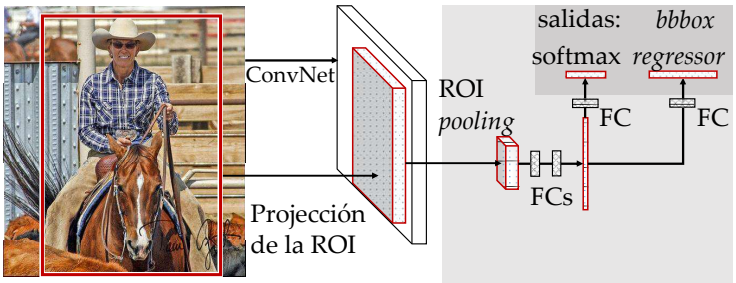


Figura 2.20. Arquitectura de Fast R-CNN [46].

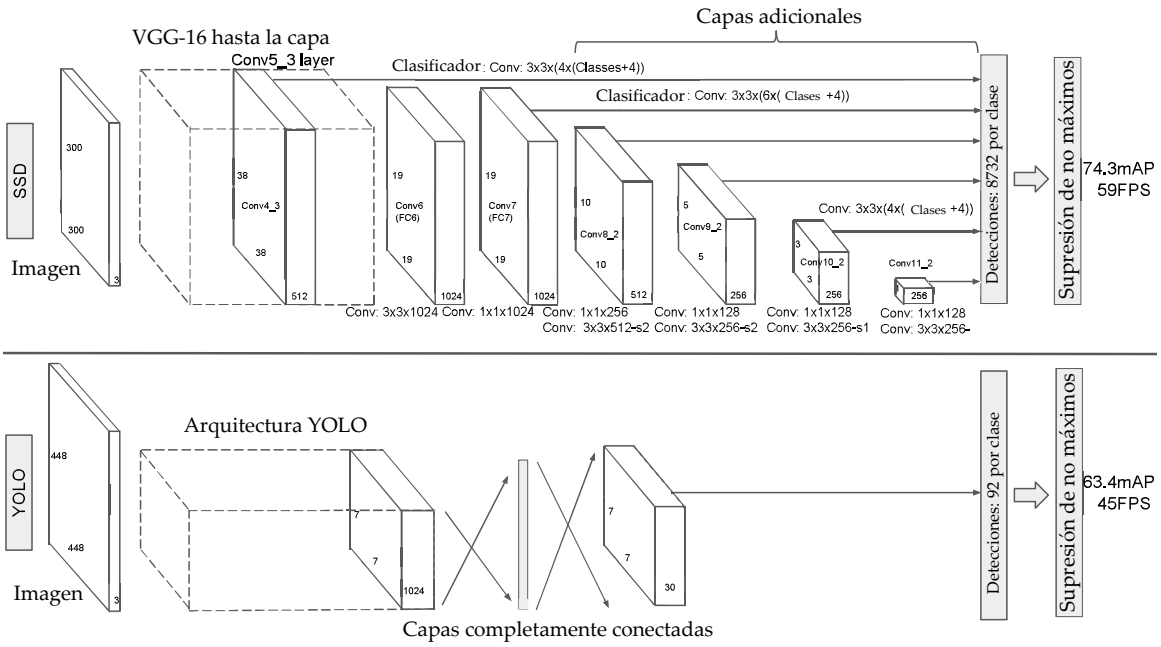


Figura 2.21. Arquitectura SSD comparada con YOLO [47].

2.6.3. SSD

El detector de objetos Single Shot Detector [47] o SSD por sus siglas en Inglés surgió para mejorar algunos de los problemas que existían en los detectores de dos etapas. Al ser un detector de una etapa, no es necesario realizar múltiples muestreos sobre imágenes o mapas de características para realizar la detección de objetos. Las mejoras principales de esta arquitectura incluyen utilizar filtros convolucionales pequeños para clasificar y redimensionar las cajas de anclaje, utilizando diferentes predictores para las detecciones con una relación de aspecto diferentes, y aplicar estos filtros a múltiples mapas de características en las etapas posteriores de la red para realizar detecciones a múltiples escalas [47].

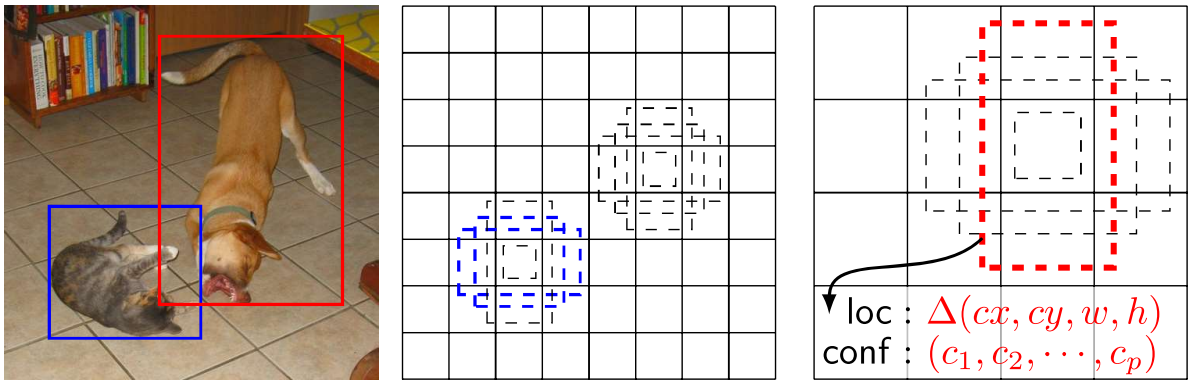
La arquitectura SSD se basa en una red convolucional de propagación hacia delante que produce una colección fija de cuadros delimitadores y certidumbres para los objetos dentro de esos cuadros y posteriormente se utiliza una etapa de supresión de no máximos para determinar los cuadros que están por arriba de un umbral de certidumbre. En la Figura 2.21 se muestra la arquitectura SSD comparada con la primer versión de YOLO [48]. El *backbone* de SSD utiliza las primeras 5 ca-

pas convolucionales de la arquitectura VGG-16 (sin incluir las capas de clasificación), las cuales utilizan filtros con un tamaño de 3×3, y alternan capas *pooling* cada 2 capas convolucionales (excepto por la capa 5). Posteriormente, agregan capas convolucionales auxiliares para generar mapas de características a diferentes escalas y, de esta manera, poder realizar detecciones en múltiples escalas. Otra característica clave de SSD es el hecho de que utiliza capas convolucionales para realizar las detecciones. Estas capas convolucionales producen una certidumbre para las categorías y las correcciones a las cajas de anclaje. Los tamaños y las relaciones de aspecto de las cajas de anclaje, o cuadros delimitadores predeterminados, se obtienen tomando en cuenta que los mapas de características responden de manera diferente a objetos de diferentes tamaños. Por ejemplo, suponiendo que se requieren m mapas para la predicción, entonces los tamaños predeterminados para cada mapa de característica se obtiene con:

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m-1}(k-1), k \in [1,m] \tag{2.16}$$

donde s_{min} es 0.2 y s_{max} es 0.9, lo que significa que la capa menos profunda utiliza una escala de 0.2 y la capa mas profunda una escala de 0.9. Por otro lado, para las relaciones de aspecto se considera el conjunto $a_r \in \{1,2,3,\frac{1}{2},\frac{1}{3}\}$. Utilizando estos aspectos,se puede calcular el ancho ($w_k^a = s_k\sqrt{a_r}$) y la altura ($h_k^a = s_{h/\sqrt{a_r}}$) de cada cuadro predeterminado. Además, para la relación de aspecto igual a 1, se agrega otro cuadro con una escala de $s_k\prime = \sqrt{s_k s_{k+1}}$, resultando en un total de 6 cuadros predeterminados por cada mapa de características.

Como se observa en la Figura 2.21, el modelo SSD utiliza más capas después de su respectivo *backbone*, las cuales se encargan de predecir los *offsets* de cada cuadro predeterminado y sus respectivas certidumbres para cada clase. Comparado con la primer versión de YOLO, SSD utiliza imágenes de entrada con tamaños de 300×300 lo que hace que sea más veloz. Por otra parte, aún con imágenes de entrada más pequeñas, SSD logra tener mejores resultados sobre el conjunto de datos VOC2007.



(a) **(b)** **(c)**

Figura 2.22. Arquitectura SSD: (**a**) etiquetado de las imágenes, (**b**) mapa de características de 8x8 con cuadros de anclaje, (**c**) mapa de características de 4x4 con cajas de anclaje [47].

En la Figura 2.22 se muestra un ejemplo de una imagen etiquetada necesaria para el entrenamiento. Esta imagen se pasará por múltiples capas convolucionales para obtener mapas de características de diferentes tamaños. Para cada mapa de características se evaluarán las correcciones ($\Delta(cx, cy, w, h)$) necesarias para enmarcar de mejor manera al objeto y las certidumbres para todas las categorías de objetos ($(c_1, c_2, \ldots, c_p)$).

2.6.4. YOLOv4

La arquitectura YOLO[48] fue propuesta por Joseph Redmon en un articulo del 2016. Actualmente, se encuentra en su cuarta versión (YOLOv4) [49] y su principal ventaja es que puede correr en tiempo real, manteniendo altas precisiones. Para obtener un alto rendimiento con YOLOv4, se utilizan procedimientos adicionales que el autor categoriza como *Bag-of-Freebies* y *Bag-of-Specials*.

Los procedimientos que conforman el *Bag-of-Freebies* son aquellos que buscan mejorar la exactitud sin incrementar el costo de inferencia. Estos procedimientos hacen cambios en la forma que se entrena la red o bien buscan mejorar el costo de entrenamiento. Uno de los principales métodos para realizar esto es utilizando *data augmentation* o bien, un aumento de datos para incrementar la variabilidad de las imágenes de entrada de tal manera que el modelo adquiera mayor robustez a

imágenes adquiridas en diferentes entornos o ambientes. Los métodos que se utilizan para realizar el aumento de datos incluyen distorsiones fotométricas y geométricas. Para generar distorsiones fotométricas se hacen cambios en el brillo, el contraste, el matiz, la saturación y/o el ruido en la imagen. Por otra parte, para generar distorsiones geométricas se hacen cambios de escala y recortes o rotaciones de la imagen.

La categoría de *Bag-of-Specials* se refiere a los módulos y métodos de post-procesamiento que pueden mejorar significativamente la exactitud. De manera general, estos módulos buscan mejorar diferentes atributos de los modelos como por ejemplo incrementar el campo receptivo, introduciendo mecanismos de atención, o mejorando la capacidad de integrar diferentes mapas de características. Por otra parte, los métodos de post-procesamiento se utilizan para examinar las predicciones del modelo. Los modulos más comunes que pueden mejorar el campo receptivo incluyen Spatial Pyramid Pooling (SPP) [1], Atrous Spatial Pyramid Pooling (ASPP) [50] y Receptive Field Block (RFB) [51].

Considerando la arquitectura general de un detector de objetos (ver Figura 2.16), YOLOv4 utiliza a la arquitectura CSPDarknet53 como *backbone*, la cual consiste de 29 capas convolucionales con filtros de 3×3, un campo receptivo de 725×725 y 27.6 millones de parámetros. Posteriormente, utiliza un cuello SPP (se muestra en la Figura 2.23) para incrementar el campo receptivo y separar las características más significativas sin afectar la velocidad del algoritmo. Este cuello se implementa considerando el trabajo de [1], en donde se concatenan mapas de características de diferentes tamaños, pero con el mismo numero de canales para obtener una representación de tamaño fijo. De esta manera, se puede utilizar la representación final para una capa complemente conectada (como en el trabajo de [1]). Debido a que la salida es un vector de una dimensión, no se puede utilizar para redes convolucionales. Por lo tanto, [52] mejora el modulo utilizando *up-sampling*, de tal manera que se puedan unir los mapas de características y se puedan utilizar en las siguientes capas convolucionales.

Posteriormente, se utiliza bloques de agregaron basados en la arquitectura PANet de [53] para combinar las características obtenidas en diferentes etapas del *backbone*. Normalmente, PANet

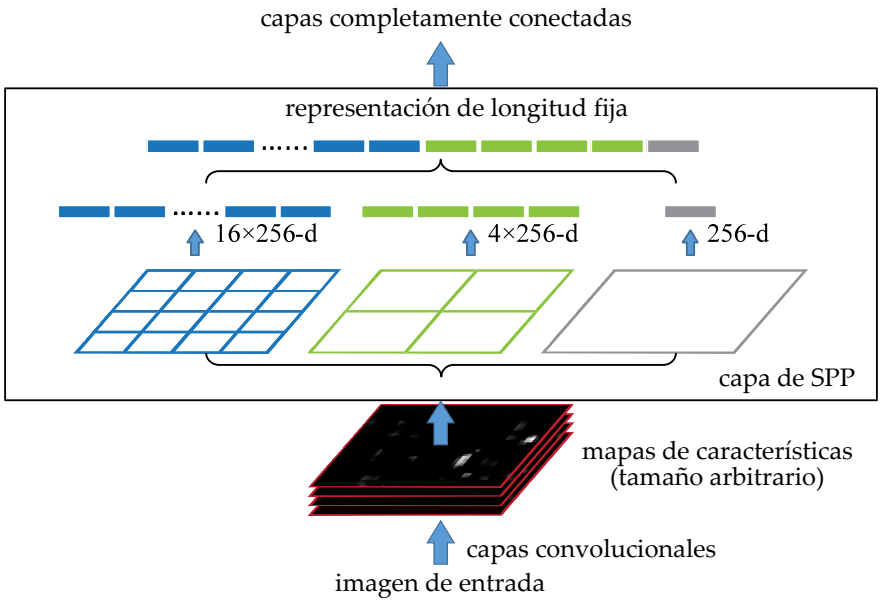


Figura 2.23. Estructura de Spatial Pyramid Pooling (SPP) propuesta por [1].

suma las capas vecinas (de la mismas dimensiones) para posteriormente realizar predicciones. En comparación, YOLOv4 realiza una operación de concatenación y esto permite mejorar la exactitud de las predicciones [49].

Finalmente, para realizar las predicciones, utiliza una cabeza YOLOv3 [52]. Esta cabeza realiza predicciones en tres diferentes escalas considerando los mapas de características que fueron concatenados utilizando los módulos de PAN. Utilizando capas convolucionales, la cabeza YOLOv3 genera un tensor de 3 dimensiones que contiene el cuadro delimitador, una certidumbre de si existe o no un objeto y las certidumbres para cada clase.

2.6.5. Comparación

La principal similaridad de los detectores esta en el hecho de que todos utilizan cuadros de anclaje para realizar la detección. De esta manera, la red solamente predice las correcciones necesarias para enmarcar al objeto de mejor manera. Por otra parte, SSD y YOLOv4 son detectores de una etapa, por lo tanto generan múltiples predicciones y se seleccionan las de mayor certidumbre para generar los resultados. En comparación, Faster R-CNN es un detector de dos etapas y primero

utiliza una predicción densa para proponer las posibles ubicaciones de los objetos y posteriormente realiza una predicción escasa para determinar exactamente los objetos que existen y corregir las dimensiones de los cuadros de anclaje. Debido a las dos etapas de Faster R-CNN, este tiende a ser más lento que los de una etapa. Además, los autores de Faster R-CNN optan por no realizar detecciones sobre mapas de características de múltiples tamaños (como se hace en SSD y YOLO) debido a el incremento en complejidad y costo computacional.

Los tres detectores de objetos fueron pre-entrenados utilizando la base de datos COCO, y aunque todos utilizan algún método de aumento de datos, YOLOv4 se enfoca en su uso para mejorar significativamente su desempeño en comparación con los detectores en el estado del arte. Tomando esto en cuenta, para entrenar YOLOv4 propone dos métodos principales: cortar y mezclar imágenes y generar mosaicos utilizando múltiples imágenes.

Por otra parte, aunque las tres arquitecturas utilizan cajas de anclaje, los métodos para determinar estas cajas es muy diferente para cada arquitectura. Primero, para Faster R-CNN, se consideran 9 cajas de anclaje por cada ventana deslizante. Para estas cajas se consideran 3 diferentes tamaños y 3 diferentes relaciones de aspecto para generar una pirámide de cajas de anclaje. Al hacer esto, los autores declaran que las cajas de anclaje son invariantes a la traslación y que la pirámide de cajas de anclaje ataca el problema de poder detectar objetos de múltiples tamaños [44].

Para obtener las cajas de anclaje, YOLOv4 primero utiliza el método de agrupación *K-means* sobre las cuadros delimitadores del conjunto de entrenamiento. Posteriormente, seleccionan 9 grupos y se ordenan dependiendo de las escalas. En comparación, SSD utiliza la ecuación 2.16 para determinar los tamaños de las cajas de anclaje. Estos tamaños son dependientes de las escalas predeterminadas establecidas por el autor.

Considerando las diferencias y similaridades, una de las principales desventajas de Faster R-CNN es que tiende a ser computacionalmente costoso, aún con *hardware* de gama alta. Esto lo hace inadecuado para su uso en computadoras sin GPU, como es el caso del robot de rescate que se utiliza en esta tesis.

2.6.6. Métricas de evaluación

Para evaluar el desempeño de los detectores de objetos se pueden utilizar distintas métricas. En este trabajo se utilizó la precisión, el *recall*, el puntaje F1, la media del promedio de precisión (mAP) y la intersección sobre unión (IoU).

Para definir las primeras tres métricas, se considera un clasificador binario que discrimina entre dos clases diferentes. A la clase que deseamos clasificar se puede denominar clase positiva (P) y a los demás ejemplos se les asignara la clase negativa (N). Al utilizar este tipo de clasificador para generar predicciones sobre un conjuto de datos previamente etiquetados, solo se pueden dar 4 posibilidades [36]:

- Verdaderos positivos (TP, por sus siglas en ingés): Los ejemplos de la clase positiva que el clasificador puede clasificar correctamente.
- Falsos positivos (TN, por sus siglas en ingés): Los ejemplos que no pertenecen a la clase positiva, pero el clasificador los clasifica como si sí lo fueran.
- Falsos negativos (FN, por sus siglas en inglés): Los ejemplos que no pertenecen a la clase negativa, pero el clasificador los clasifica como si sí li fueran.
- Verdaderos negativos (TN, por sus siglas en inglés) Los ejemplos que pertenecen a la clase negativa y el clasificador los clasifica correctamente.

Los cuatro casos mencionados se pueden representar en una matriz conocida como matriz de confusión. Para el clasificador binario mencionado anteriormente, la matriz tendría un tamaño de 2×2 como se muestra en la Figura 2.24. Para un problema de clasificación de K clases, la matriz de confusión tendría una dimensión de de $K \times K$.

Utilizando la matriz de confusión, se puede obtener diferentes métricas de evaluación que se utilizan comúnmente para comparar diferentes modelos y determinar su desempeño. La exactitud (*accuracy* en Inglés) es la métrica más utilizada para determinar el rendimiento de un modelo de

		Predicción: P	Predicción: N
Clase real	P	TP	FN
	N	FP	TN

Figura 2.24. Matriz de confusión para un clasificador binario [36].

aprendizaje y se calcula considerando todos los elementos de la matriz de confusión:

$$exactitud = \frac{TP+TN}{TP+TN+FP+FN} \tag{2.17}$$

Si consideramos que n es el numero total de ejemplos del conjunto que se utiliza para evaluar el modelo, entonces se tiene que:

$$exactitud = \frac{TP+TN}{n} \tag{2.18}$$

Ahora bien, considerando un problema con k clases diferentes, entonces la exactitud se obtiene al sumar todos los aciertos que obtiene de cada clase y se divide por el numero total de ejemplos n del conjunto de datos.

Tomando en cuenta lo mencionado anteriormente, para realizar la evaluación de los detectores de objetos existen dos medidas que se utilizan muy comúnmente, precisión y exhaustividad (*recall* en Inglés). Considerando la matriz de confusión, la precisión se define como:

$$precision = \frac{TP}{TP+FP} \tag{2.19}$$

Por otra parte, el *recall* esta definido como:

$$recall = \frac{TP}{TP+FN} \tag{2.20}$$

Considerando estas definiciones, puede observar que la precisión se centra en las predicciones positivas, mientras que el *recall* se enfoca en la completitud de los resultados [36]. La precisión penaliza los falsos positivos y el *recall* los falsos negativos.

Durante el proceso de selección de hiperparámetros, se tiene que tomar en cuenta el hecho de que cuando se intenta mejorar la precisión, se hace a costa de reducir el *recall* y vise versa. Tomando en cuenta lo dicho, se pueden combinar ambas métricas para obtener una métrica que penaliza el hecho de que una de las dos (precisión o *recall*) tengan un valor bajo. A esta métrica se le conoce como la media F o *F-score* y esta definida como la media armónica:

$$F = \frac{2 * precision * recall}{precision + recall} \tag{2.21}$$

O bien, en términos de las entradas de la matriz de confusión:

$$F = \frac{2TP}{2TP + FP + FN} \tag{2.22}$$

Para el caso de los detectores de objetos, también existen dos definiciones adicionales para la precisión y el *recall* (ver Figura 2.25). Estos términos, cuantifican la habilidad del modelo para enmarcar el objeto utilizando un cuadro delimitador. Primero, la precisión se define como la relación entre la intersección del cuadro delimitador verdadero (el objeto) y el cuadro detectado entre el cuadro delimitador verdadero. De manera similar, se obtiene el *recall*, pero ahora se utiliza el cuadro delimitador detectado en el denominador. Cabe aclarar que estas métricas no se utilizan para el calculo del mAP, sino que son métricas auxiliares que permiten definir el comportamiento de un cuadro delimitador.

Adicionalmente, para determinar si una detección (un cuadro delimitador) es un verdadero positivo o un falso positivo se utiliza una métrica conocida como intersección sobre unión o IoU por sus siglas en Inglés. La IoU es una relación de la intersección del cuadro delimitador verdadero (el objeto) y el cuadro detectado entre la unión de los mismos (ver Figura 2.25). Si valor de IoU es mayor a un umbral IoU_{umbral}, entonces es un verdadero positivo (TP), de lo contrario es un falso

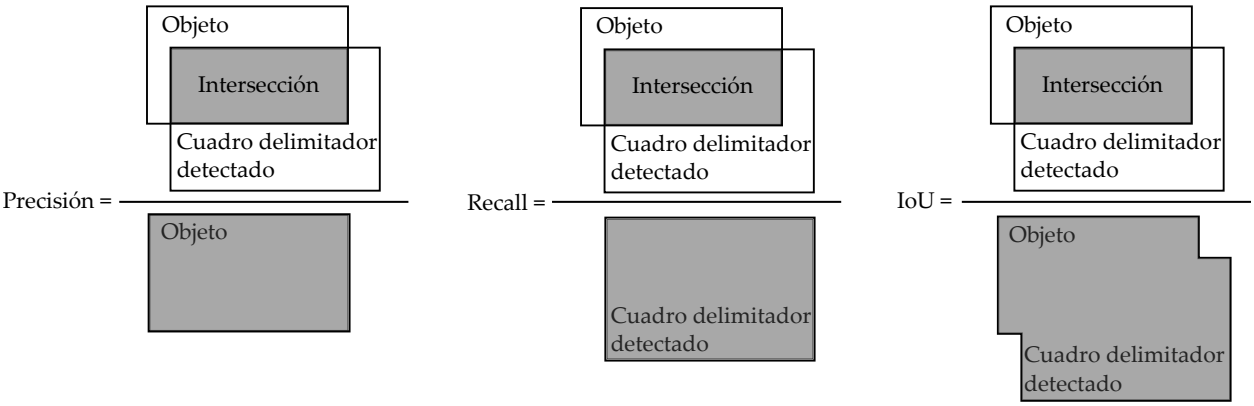


Figura 2.25. Representaciones de precisión, *recall* e intersección sobre unión [54].

positivo (FP). Además, si no se genera una predicción para un cuadro delimitador verdadero, entonces a ese caso se le conoce como falso negativo (FN). Entonces, considerando las definiciones hechas para los detectores de objetos, se puede volver a difinir precisión y *recall* como:

$$precision = \frac{TP}{TP+FP} = \frac{TP}{\text{todos los cuadros detectados}} \tag{2.23}$$

$$recall = \frac{TP}{TP+FN} = \frac{TP}{\text{todos los cuadros verdaderos}} \tag{2.24}$$

Precisión media promedio (mAP)

Tomando en cuenta las métricas mencionadas anteriormente, se puede definir la forma en que se determina el mAP. Esta métrica se obtiene a partir del promedio de la precisión (AP, por sus siglas en Inglés), la cual a su vez se obtiene utilizando la curva de precisión contra *recall* (curva PR). Existen múltiples variantes para obtener al AP, pero solamente se hará la descripción matemática de la que se utilizó para la evaluación de los modelos.

La curva PR muestra la relación entre la precisión y el *recall* para diferentes valores de certidumbre asociados a los cuadros delimitadores generados por el detector. Si la certidumbre de un detector genera pocos FP, entonces la precisión sera alta, pero existirá un alto número de FN y, por lo tanto, un *recall* bajo. De manera contraria, si se aceptan más positivos, entonces el *recall*

incrementará, pero también lo harían los FP y por lo tanto disminuiría la precisión [55].

Un detector de objectos con un alto desempeño debería encontrar todos los cuadros delimitadores verdaderos e identificar solamente los objetos relevantes. O bien, como se menciono anteriormente, debería obtener una precisión y *recall* altos, aún cuando los valores de certidumbre varían. Dicho lo anterior, si el área debajo (AUC, por sus siglas en Inglés) de la curva PR es grande, entonces eso indicaría una precisión y *recall* altos.

Normalmente, la curva PR tiende a tener una forma en zigzag y hace que el calculo del AUC sea más complejo. Para solucionar este inconveniente se utilizan dos métodos que eliminan el comportamiento en zigzag, la interpolación de 11 puntos y la interpolación de todos los puntos. La interpolación de 11 puntos se utiliza para obtener una curva sin zigzags utilizando el promedio de las precisiones máximas en 11 valores de *recall* igualmente espaciados $[0, 0.1, 0.2, \ldots, 0.9, 1]$, y se define como:

$$AP_{11} = \frac{1}{11} \sum_{R \in \{0, 0.1, \ldots, 0.9, 1\}} P_{interp}(R) \tag{2.25}$$

donde

$$P_{interpo}(R) = \max_{\tilde{R}:\tilde{R} \geq R} P(\tilde{R}) \tag{2.26}$$

En esta definición de AP, solamente se considera la precisión máxima $P_{interp}(R)$ correspondiente a el *recall* que es mayor a R.

Para el caso de la interpolación que utiliza todos los puntos se tiene que:

$$AP_{all} = \sum_{n} (R_{n+1} - R_n) P_{interp}(R_{n+1}) \tag{2.27}$$

donde

$$P_{interp}(R_{n+1}) = \max_{\tilde{R}:\tilde{R} \geq R} P(\tilde{R}) \tag{2.28}$$

En este caso, en lugar de utilizar la precisión solamente en algunos puntos, la AP se obtiene al interpolar la precisión en cada nivel, tomando la precisión máxima que corresponde al valor de *recall* mayor o igual a R_{n+1}.

Las APs consideradas anteriormente, consideran un caso donde solamente existe una clase a detectar. Por lo tanto, para poder evaluar el detector correctamente (sobre todas las clases para las cuales esta diseñado) se utiliza la media de la AP o mAP. La mAP es simplemente la AP promedio sobre todas las clases:

$$mAP = \frac{1}{N}\sum_{i=1}^{N} AP_i \tag{2.29}$$

donde AP_i es la AP en la i-ésima clase y N es el numero total de clases para las cuales se esta evaluando [55].

2.7. Lógica Difusa

Los robots autónomos necesitan interpretar y utilizar información incompleta e imprecisa obtenida de sensores tales como cámaras o LIDARs. La Teoría de Conjuntos Difusos provee un calculo sistemático para lidiar con este tipo de información de manera lingüística, utilizando etiquetas lingüísticas estipuladas por funciones de membresia. Además, utilizando reglas difusas del tipo si-entonces (if-then en ingles), se puede modelar la experiencia humana en aplicaciones específicas [56].

2.7.1. Conjuntos difusos

En comparación con los conjuntos clásicos, un conjunto difuso no tiene una frontera estricta. Esto significa que la transición entre la “pertenencia” y “no pertenencia” es gradual y esta transición suave esta caracterizada por funciones de membresia. Un conjunto difuso A, cuyo universo es X, esta definido por una colección de elementos $x \in X$ tal que cada x pertenece en cierto grado al

conjunto A [56, 57].

Definición 1 *Conjuntos difusos y funciones de membresia*

Si X es una colección de objetos denotados genéricamente por x, entonces un conjunto difuso A en X esta definido como un conjunto de pares ordenados:

$$A = \{(x, \mu_A(x)) | x \in X)\} \tag{2.30}$$

donde $\mu_A(x)$ es conocida como función de membresia (MF por sus siglas en inglés) del conjunto A. La MF mapea cada elemento del universo de discurso X a un valor de membresia entre 0 y 1. □

2.7.2. Principio de extensión y relaciones difusas

El principio de extensión [58, 59, 56] es un concepto básico de la teoría de conjuntos difusos que provee un procedimiento general para extender las expresiones matemáticas, por ejemplo operaciones algebraicas, en dominios discretos o *crisp* a dominios difusos. Este procedimiento generaliza un mapeo común de punto a punto de una función $f(\cdot)$ a un mapeo entre conjuntos difusos.

Por otra parte, las relaciones difusas binarias son conjuntos difusos en $X \times Y$ que mapean cada elemento en $X \times Y$ a un grado de membresia entre 0 y 1. En particular, las relaciones unarias son conjuntos difusos con MFs de una dimensión, las relaciones difusas binarias son conjuntos difusos de dos dimensiones, etc. Algunos ejemplos de relaciones binarias difusas son [56]:

- x **esta cerca a** y (x y y son números)
- x **depende de** y (x y y son eventos)
- x y y **se parecen** (x y y son personas, objetos, etc.)
- **Si** (If en inglés) x es grande, **entonces**(then en inglés) y es pequeño (x es una lectura observada de algún proceso y y es la acción correspondiente)

El último ejemplo utiliza una expresión **If-then** comúnmente utilizada en sistemas de inferencia difusa (FIS), como el que se presente en este trabajo.

Las relaciones difusas se pueden combinar utilizando la operación de composición. Diferentes operaciones de composición han sido propuestas, pero la más conocida es la composición $max-min$ propuesta por Zadeh [58]. Considerando dos relaciones difusas $\mathscr{R}_1$ y $\mathscr{R}_2$ definidas en $X \times Y$ y $Y \times Z$ respectivamente, la composición $max-min$ de estas relaciones es un conjunto difuso definido por

$$\mathscr{R}_1 \circ \mathscr{R}_2 = \{[(x,z), \underset{y}{maxmin}(\mu_{\mathscr{R}_1}(x,y), \mu_{\mathscr{R}_1}(y,z))] | x \in X, y \in Y, z \in Z\} \tag{2.31}$$

2.7.3. Variables lingüísticas

Una variable lingüística se caracteriza por una quintuplica $(x, T(x), X, G, M)$ en donde x es el nombre de la variable; $T(x)$ es el conjunto de términos de x – esto es, el conjunto de sus valores o términos lingüísticos; X es el universo de discurso; G es una regla sintáctica la cual genera los términos en $T(x)$; y M es una regla semántica, la cual asocia el significado $M(A)$ con cada valor lingüístico A, donde $M(A)$ denota un conjunto difuso X [56].

2.7.4. Reglas difusas If-Then

Una regla difusa if-then, también conocida como regla difusa, implicación difusa, o proposición condicional difusa toma la forma:

$$\text{if } X \text{ is } A \text{ then } y \text{ is } B \tag{2.32}$$

donde A y B son valores lingüísticos definidos por conjuntos difusos en universos de discurso X y Y, respectivamente. Normalmente, a “x is A” se le conoce como el antecedente o premisa, mientras que a “y is B” se le conoce como la consecuencia o la conclusión. La expresión de la ecuación 2.32

también se abrevia como $A \rightarrow B$. En esencia, la expresión describe una relación entre dos variables x y y. Esto sugiere que una regla difusa este definida como una relación difusa $\mathscr{R}$ sobre el espacio del producto $X \times Y$.

De manera general, hay dos formas de interpretar una regla difusa $A \rightarrow B$. La primera, interpreta a la regla difusa como " A combinado con B " y se puede expresar como:

$$\mathscr{R} = A \rightarrow B = A \times = \int_{X \times Y} \frac{\mu_A(x) \tilde{*} \mu_B(y)}{(x,y)} \tag{2.33}$$

donde el operador $\tilde{*}$ es un operador norma T. Por otra parte, la segunda interpretación de $A \rightarrow B$ se interpreta como "A implica B ", la cual puede expresarse de 4 formas:

- Implicación material

$$\mathscr{R} = A \rightarrow B = \neg A \cup B \tag{2.34}$$

- Cálculo proposicional

$$\mathscr{R} = A \rightarrow B = \not{A} \cup (A \cap B) \tag{2.35}$$

- Cálculo proposicional extendido

$$\mathscr{R} = A \rightarrow B = (\neg A \cap \neg B) \cup B \tag{2.36}$$

- Generalización del modus ponens

$$\mu_{\mathscr{R}}(x,y) = sup(c | \mu_A(x) \tilde{*} c \leq \mu_B(y) \text{ and } 0 \leq c \leq 1 \tag{2.37}$$

  donde $\tilde{*}$ es el operador norma T.

En esencia, las ultimas 4 expresiones se pueden reducir a la implicación material de la lógica bi-valuada $A \rightarrow B = \neg A \cup B$.

Finalmente, considerando las dos interpretaciones mencionadas y en varios operadores norma T y conorma T, se pueden generar diferentes métodos para calcular la relación difusa $\mathbb{R} = A \rightarrow B$. Debe notarse que $\mathbb{R}$ se puede ver como un conjunto difuso con una función de membresia de dos dimensiones:

$$\mu_{\mathscr{R}} = f(\mu_A(x), \mu_B(y)) = f(a,b) \tag{2.38}$$

donde $a = \mu_A(x)$, $b = \mu_B(y)$ y a f se le conoce como la función de implicación difusa y se encarga de transformar los grados de membresia de x en A y de y en B en áquellos grados de membresia de (x,y) en $A \rightarrow B$.

Suponiendo que se utiliza la primer interpretación, "A combinando con B" para la definición de $A \rightarrow B$ entonces se pueden generar cuatro diferentes relaciones difusas utilizando los cuatro operadores norma T más comunes:

- $$\mathscr{R}_m = A \times B = \int_{X \times Y}^{0} \frac{\mu_A(x) \wedge \mu_B(y)}{(x,y)} = f_m(a,b) = a \wedge b \text{ (Mamdani)} \tag{2.39}$$

- $$\mathscr{R}_m = A \times B = \int_{X \times Y}^{0} \frac{\mu_A(x)\mu_B(y)}{(x,y)} = f_p(a,b) = ab \text{(Larsen)} \tag{2.40}$$

- $$\mathscr{R}_m = A \times B = \int_{X \times Y}^{0} \frac{\mu_A(x) \odot \mu_B(y)}{(x,y)} = \int_{X \times Y}^{0} \frac{0 \vee (\mu_A(x) + \mu_B(y) - 1)}{(x,y)} = f_b(a,b) = 0 \vee (a+b-1) \tag{2.41}$$

- $$\mathscr{R}_m = A \times B = \int_{X \times Y}^{0} \frac{\mu_A(x) \overset{\wedge}{\cdot} \mu_B(y)}{(x,y)}, \tag{2.42}$$

o

$$f_d(a,b) = a \overset{\wedge}{\cdot} b = \begin{cases} a & \text{si b = 1} \\ b & \text{si a = 1} \\ 0 & \text{de lo contrario} \end{cases}$$

2.7.5. Sistemas de Lógica Difusa

Un sistema difuso (FLS por sus siglas en Inglés), es una plataforma de computo basada en la teoría de conjuntos difusos, reglas difusas y razonamiento difuso. Un FLS consiste de tres componentes conceptuales básicos: una base de reglas, una base de datos de términos lingüísticos y un mecanismo de razonamiento. Estos sistemas han sido utilizados en muchos campos incluyendo control automático, clasificación de datos, análisis de decisiones, sistemas expertos, robótica y reconocimiento de patrones. Además, debido a su naturaleza multidisciplinaria, este sistema se conoce por otros nombres, por ejemplo sistema de inferencia difusa (FIS por sus siglas en inglés), modelo difuso, controlador basado en lógica difusa o simplemente sistema difuso [56].

En la figura 2.26 se muestran las etapas necesarias para un sistema difuso. El proceso de inferencia difuso inicia por la fusificación de las variables de entrada. En esta etapa se se utilizan las entradas nítidas y se determina el grado en que pertenecen a cada termino lingüístico definido por su respectiva función de membresia. Posteriormente, se utiliza la base de reglas, la base de datos de términos lingüísticos y el mecanismo de razonamiento para realizar el procedimiento de inferencia. Un sistema difuso puede utilizar entradas difusas o nítidas, pero por lo general, produce salidas difusas. Sin embargo, muchas veces, se necesitan salidas nítidas y se requiere de una etapa de defusificación que se encarga de extraer el valor nítido que mejor representa al conjunto difuso.

2.7.6. Modelo de inferencia Mamdani

El modelo de inferencia Mamdani fue propuesto por Mamdani y S. Assilian en 1995. En su articulo, titulado “An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller” [60],

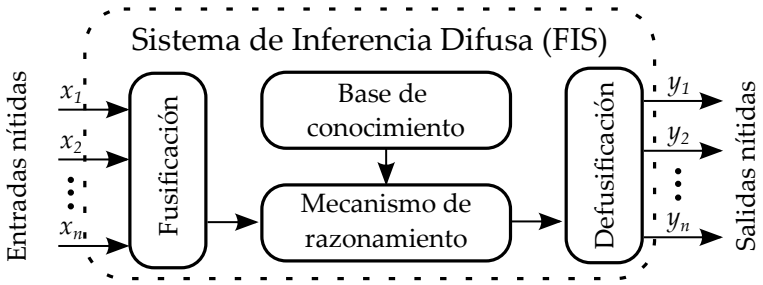


Figura 2.26. Arquitectura de un detector de objetos [43].

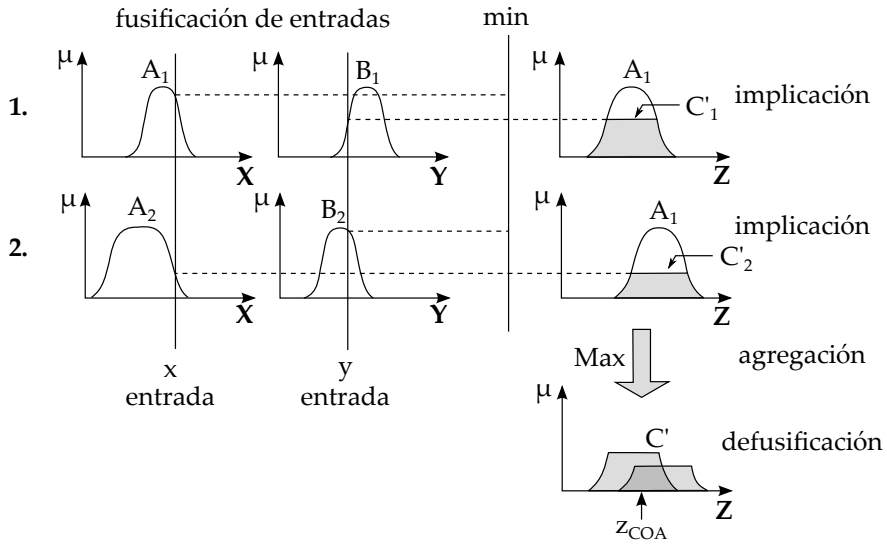


Figura 2.27. Modelo de inferencia Mamdani que utiliza *min* y *max* como operadores norma-T y conorna-T, respectivamente [56].

ellos presentan un sistema difuso con 24 reglas para el control de una máquina de vapor. Las aportaciones de este trabajo fue utilizar el operador *min* para la implicación de las premisas y también demostrar que se puede representar la experiencia humana en un conjunto de reglas difusas [57].

En la Figura 2.27 se muestra una ilustración del funcionamiento de un Sistema de Inferencia Mamdani de dos reglas [56]. Durante la etapa de fusificación, se utilizan las entradas nítidas x y y y se determina el grado en que cada entrada pertenece a un conjunto difuso dentro del universo de discurso correspondiente (en este caso X y Y).

Con la fusificación de las entradas, se determina el grado en que se satisface cada parte del antecedente para cada regla. Si el antecedente de una regla difusa contiene más de una parte,

entonces es necesario aplicar el operador el operador difuso correspondiente. En la Figura 2.27 se utiliza el operador difuso *min* para obtener el valor que representa el resultado del antecedente.

Después de la fusificación y la aplicación del operador difuso se obtiene el valor que se utiliza para realizar la implicación. Para realizar la implicación se utilizan las funciones de membresia de la variables lingüísticas de salida y el valor que se generó al aplicar el operador difuso a las entradas fusificadas. La salida de la implicación es un conjunto difuso para cada regla difusa. Estos conjuntos se combinan en un solo conjunto difuso durante la etapa de agregación. En la Figura 2.27 la etapa de agregación se realiza utilizando el operador *max*.

Finalmente, utilizando el conjunto obtenido después de la agregación, se realiza el proceso de fusificación. Esta etapa se encarga de generar un valor nítido a partir del conjunto de entrada. Algunos de los métodos para realizar la defusificación incluyen el cálculo del centroide, el calculo de la bisectriz, uso del promedio de los valores máximos del conjunto difuso o el uso del máximo más grande o el más pequeño [61].

Capítulo 3

Desarrollo

En este capítulo se presenta el proceso realizado para desarrollar el sistema de detección de escaleras. Primero, se describe la obtención del conjunto de datos, el etiquetado de imágenes y el desarrollo e implementación de la arquitectura del detector. Después, se muestran los resultados del entrenamiento y se presenta la validación del mismo. Posteriormente, se describe el controlador para la alineación autónoma con las escaleras y, finalmente, se explica la implementación del detector y el controlador en ROS.

3.1. Obtención y etiquetado de imágenes

Para la primer parte del desarrollo del detector se utilizó un conjunto de imágenes generado a partir de tres conjuntos diferentes. El primer conjunto contiene 573 imágenes de escaleras obtenidas de la base de datos MCIndoor20000. Estas imágenes fueron tomadas dentro de la Clinica de Marshfield en Wisconsin, Estados Unidos en el verano de 2017 [62]. Algunas imágenes estaban repetidas o mostraban la parte inferior de las escaleras y, a causa de esto, fueron eliminadas. Además, las imágenes no estaban etiquetadas y, por lo tanto, se realizó el etiquetado utilizando la aplicación *LabelImg* [63]. En la Figura 3.1 se muestra una imagen etiquetada de este conjunto.

El segundo conjunto de datos consiste de imágenes descargadas de *Google Images* [64] utili-

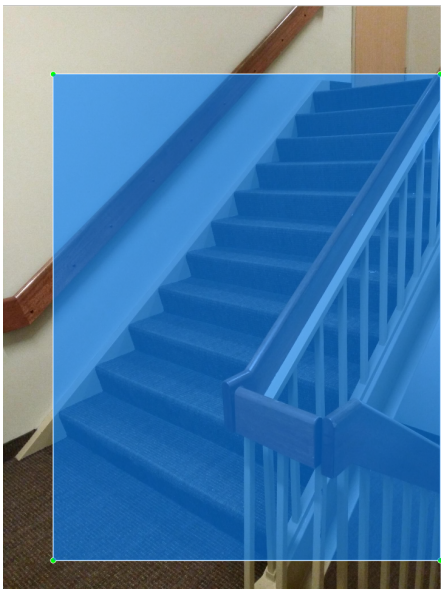


Figura 3.1. Imagen tomada de la base de datos MCIndoor20000 [62].

zando un método conocido como *image scraping* [65]. Con este método se obtuvo un conjunto de 585 imágenes y se etiquetaron, manualmente, utilizando la aplicación *LabelImg*. En la Figura 3.3 se muestra una imagen etiquetada de este conjunto.

Finalmente, el tercer conjunto de imágenes se obtuvo de la base de datos *Open Images* [66] y contiene 4668 imágenes de escaleras previamente etiquetadas. El conjunto de imágenes se obtuvo utilizando *image scraping* en Flickr[1] y fueron etiquetadas manualmente por profesionales [67]. En la figura 3.2 se muestra un ejemplo de este conjunto.

Los tres conjuntos se unieron para obtener un total de 5826 imágenes etiquetadas. De este conjunto, se utilizó el 70 % para entrenamiento, el 20 % para validación y el 10 % para pruebas. La base de datos generada utiliza el formato de etiquetado YOLO. Al utilizar este formato se genera un archivo de texto (con extensión *.txt*) el cual contiene información sobre las clases de los objectos dentro de la imagen y sus ubicaciones (dadas por cuadros delimitadores).

Cabe mencionar que, para utilizar el conjunto de datos con la API de Detección de Objetos de TensorFlow [68], se decidió convertir los conjuntos de datos al formato TFRecord. Para la conversión, primero, fue necesario mover las imágenes a un directorio y las etiquetas (archivos

[1] Servicio de almacenamiento de imágenes (`flickr.com`).

Figura 3.2. Imagen tomada de la base de datos Open Images [67].

Figura 3.3. Imagen tomada de Google Images [64].

de texto) a otro. Los dos directorios se utilizaron como parámetros del programa encargado de convertir el conjunto de datos al formato TFRecord [69].

3.2. Desarrollo e implementación de los detectores

Para la implementación de los 3 primeros detectores se utilizó *transfer learning*, tomando modelos preentrenados y realizando un reentrenamiento con el conjunto de imágenes de escaleras. Los 3 modelos de detección utilizados son:

- YOLOv4-tiny [43].
- Faster R-CNN ResNet50 [70], [71].
- SSD ResNet50 [47], [71].

El primer modelo que se entrenó es YOLOv4-tiny. Este detector recibe imágenes de entrada con un tamaño de 416x416 pixeles y utiliza un *backbone* (ver Figura 2.16) CSPDarknet53 comprimido que contiene 9 capas convolucionales (en lugar de 29 en el modelo de tamaño completo) con *kernels* de 3x3. Posteriormente, utiliza dos módulos adicionales (*neck*), Spatial Pyramid Pooling (SPP) y Path Aggregation Network (PAN). El primer módulo, SPP, ayuda a incrementar el campo receptivo y separa las características más significativas sin afectar la velocidad del algoritmo. Después, el modulo PAN se encarga de combinar las características obtenidas de la CNN (en este caso la CSPDarknet53 comprimida). Finalmente, el modelo realiza una predicción de las ubicaciones y clases de los objetos en la imagen. El detector YOLOv4-tiny se implementó en el *framework* para redes neuronales Darknet [72] y se entrenó utilizando Google Colaboratory[2] (Colab).

Los modelos Faster R-CNN ResNet50 y SSD ResNet50 se obtuvieron de la colección de modelos preentrenados disponibles en la API de Detección de Objetos de Tensorflow. Ambos modelos utilizan la arquitectura ResNet50 [71] como *backbone*. ResNet50 consiste de 50 capas, de las cuales, 49 son capas convolucionales y 1 capa es completamente conectada. ResNet50 introduce el concepto de aprendizaje residual o *residual learning*, y se implementa utilizando conexiones de corto circuito. Este tipo de conexión toma la salida de una capa convolucional y la suma a la salida de otra capa convolucional, pero saltándose cierto número de capas. Con estas conexiones se logra implementar CNN más profundas, sin incrementar el número de parámetros y mejorando la precisión del detector.

La diferencia principal entre estos detectores consiste en el tipo de predicción que realizan, Faster R-CNN hace una predicción densa (*dense prediction* en inglés) y una predicción escasa (*sparse prediction* en inglés) mientras que SSD y YOLO solo utilizan una predicción densa. Faster R-CNN

[2]Entorno de máquinas virtuales para crear y ejecutar programas en Python (`colab.research.google.com`).

utiliza una red para propuestas de regiones (RPN por sus siglas en inglés), la cual se encarga de proponer las regiones donde pueden existir objetos, y posteriormente, utiliza un clasificador para determinar el objeto en la región o cuadro delimitador. Por otro lado, el detector SSD funciona de manera similar al detector YOLO, debido a que implementa la detección utilizando una sola CNN, la cual genera las detecciones y determina a qué clase pertenece el objeto. A causa de esto, YOLO y SSD tienden a ser más veloces que Faster R-CNN debido a que solamente tienen que "ver" la imagen una vez para generar predicciones.

La implementación Faster R-CNN y SSD se realizó utilizando Docker[3] en una máquina local. Con Docker, se generó una imagen que incluye todas las librerías necesarias para entrenar y ejecutar los detectores de objetos.

3.3. Entrenamiento de los detectores de escaleras

Para entrenar al detector YOLOv4-tiny se utilizó una GPU de Nvidia Modelo T4 y el entrenamiento se tardó aproximadamente 60 minutos (con 2000 iteraciones). Por otra parte, para los modelos entrenados con la API de Detección de Objetos de Tensorflow se utilizó una GPU de Nvida Modelo RTX 2070 Max-Q, en la cual, los modelos se tardaron 90 y 52 minutos en entrenar, respectivamente.

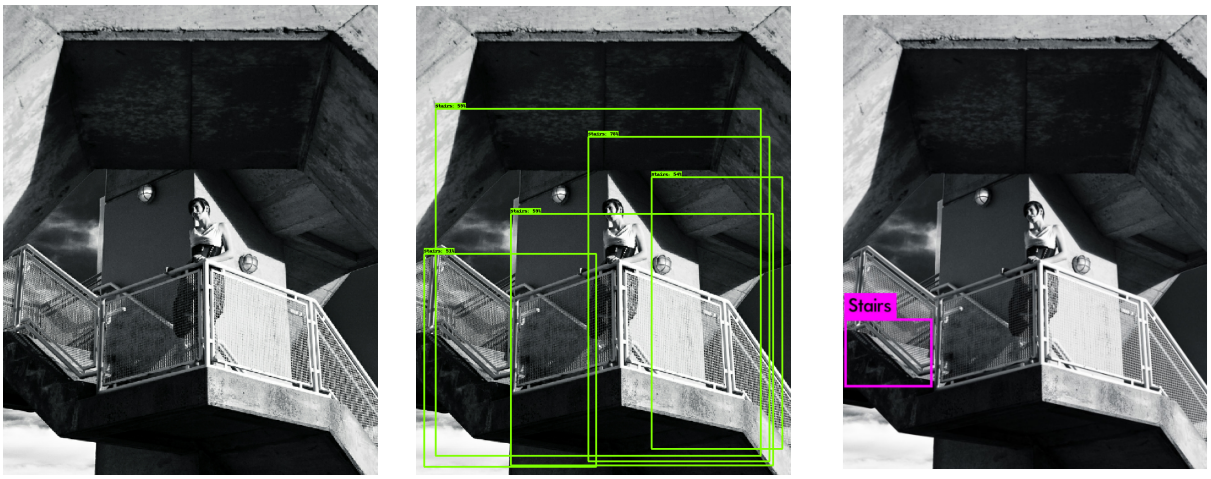
Los tres modelos se inicializaron utilizando los pesos de los modelos preentrenados con la base de datos Common Objects in Context (COCO) [74]. En la tabla 3.1 se muestra el mAP@0.50IoU de cada clasificador sobre el conjunto de prueba. De estos resultados se observa que YOLOv4-tiny tuvo el mejor desempeño con un mAP@0.5IoU de 66.20 %.

Para evaluar los modelos entrenados se utilizaron 583 imágenes de prueba. En las Figuras 3.4, 3.5, 3.6 y 3.7 se muestran algunas imágenes de escaleras y las predicciones obtenidas por cada

[3]Plataforma que permite empaquetar software en unidades estandarizadas llamadas contenedores [73].

Tabla 3.1: Tiempos de entrenamiento y mAP@0.5IoU.

Modelo	**Tiempo de Entrenamiento [min]**	mAP@0.5
YOLOv4-tiny, 2000 iteraciones	60	63.08 %
YOLOv4-tiny, 4000 iteraciones	100	66.20 %
Faster-RCNN ResNet50	90	44.59 %
SSD ResNet50	52	1.15 %

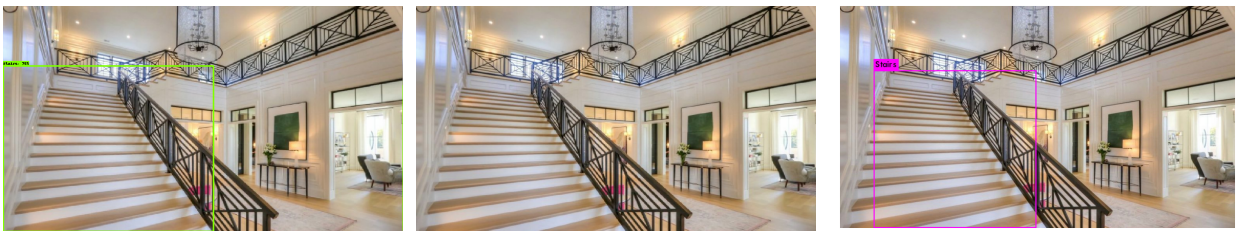


(a) Faster-RCNN ResNet50 **(b)** SSD ResNet50 **(c)** YOLOv4-tiny

Figura 3.4. Detección de escaleras en imagen de prueba.

(a) Faster R-CNN ResNet50 **(b)** SSD ResNet50 **(c)** YOLOv4-tiny

Figura 3.5. Detección de escaleras en imagen de prueba.



(a) Faster R-CNN ResNet50 **(b)** SSD ResNet50 **(c)** YOLOv4-tiny

Figura 3.6. Detección de escaleras en imagen de prueba.

(a) Faster R-CNN ResNet50 **(b)** SSD ResNet50 **(c)** YOLOv4-tiny

Figura 3.7. Detección de escaleras en imagen de prueba.

detector. De las imágenes se observa que YOLOv4-tiny (con 4000 iteraciones) genera mejores localizaciones de las escaleras, en comparación con los otros detectores, y además logra detectar la escalera que se encuentra ocluida por un barandal con malla metálica (ver Figura 3.4).

Por otra parte, el detector SSD ResNet50 obtuvo el peor desempeño con un mAP@05IoU de 1.15 %. De las imágenes de prueba se observó que el detector genera falsos positivos en zonas con ciertos patrones tales como la malla metálica del barandal (Figura 3.4b) o en espacios con aglomeraciones de objetos (Figura 3.5b). Considerando los resultados obtenidos en esta etapa y los resultados de [75] se decidió utilizar el detector YOLOv4-tiny para este trabajo y optimizarlo utilizando las recomendaciones propuestas en [54].

3.3.1. Optimización del detector de escaleras basado en YOLOv4-tiny

Para mejorar la detección, en [54] se presentan las siguientes recomendaciones:

- Incrementar el número de iteraciones, pero evitando el sobreajuste.
- Seleccionar el número apropiado de subdivisiones del lote.
- Utilizar tres etapas de detección (tres capas YOLO) para detectar objetos pequeños y grandes.

- Modificar el parámetro *random*, el cual determina si se cambiará el tamaño de la red de entrada y salida de manera aleatoria (cada 10 iteraciones) durante el entrenamiento.
- Utilizar imágenes sin escaleras (imágenes negativas) durante el entrenamiento.

En las tablas 3.2 y 3.3 se muestran las configuraciones utilizadas y los resultados obtenidos sobre el conjunto de prueba. Tomando la prueba 1 como referencia se realizaron distintos cambios en la arquitectura, las configuraciones de entrenamiento y las imágenes utilizadas para mejorar significativamente el modelo inicial. Cabe mencionar que las pruebas 2 y 3 utilizan diferentes arquitecturas para poder comparar los desempeños. La prueba 2 utiliza el modelo de tamaño completo YOLOv4, mientras que la prueba 3 utiliza la versión comprimida de YOLOv3, YOLOv3-tiny. Los resultados muestran que aunque las versiones comprimidas de YOLOv4 no logran alcanzar el desempeño de la versión completa, si se logra mejorar el desempeño inicial utilizando los métodos especificados en [54].

Para mejorar el modelo inicial, primero se deshabilitó el parámetro *random* y se observó un incremento en el mAP obtenido sobre el conjunto de imágenes de prueba (prueba 1 comparada con prueba 5). Tomando esto en cuenta, se decidió continuar utilizando esta configuración (*random*=0) para el resto de los detectores. Al agregar las imágenes negativas durante el entrenamiento (pruebas 4 comparada con prueba 6) se observó una disminución en el mAP@0.5 IoU, pero un incrementó considerable en el IoU promedio. En la prueba 7, también se incrementó el número total de iteraciones y se disminuyó el número de subdivisiones del lote. Con esta configuración fue posible obtener un mAP@0.5 de 70.93 % un IoU promedio de 61.36 % y una precisión y recall de 0.81 y 0.62 respectivamente. Finalmente, se decidió agregar 1 capa más de detección al modelo (3 capas YOLO en total) para mejorar la detección en objetos de mayor tamaño (pruebas 8 y 9) y también recalcular las cajas de anclaje (prueba 8).

Considerando los cambios y los resultados de las tablas 3.2 y 3.3 se decidió utilizar los detectores 7, 8 y 9 (obtenidos en las pruebas 7, 8 y 9 respectivamente) para probar los *pipelines* implementados en ROS. Las principal diferencia de estas arquitecturas es el hecho de que los detectores 8 y 9 utilizan 3 capas de detección en lugar de 2. Además, el detector 8 utiliza cajas de anclaje obtenidas al aplicar un algoritmo *k-means* a los cuadros delimitadores correspondientes a el conjunto de imágenes de entrenamiento. Los tamaños cajas de anclaje obtenidas utilizando este método son: (55, 47), (88,118), (214, 73), (171,176), (115,286), (330,149), (230,309), (366,242), (373,376), donde cada par representa el ancho y alto de las cajas de anclaje. En comparación, el detector 9 utiliza las cajas de anclaje especificadas en [54], las cuales están dadas por: (12, 16) , (19, 36), (40, 28), (36, 75) , (76, 55), (72, 146), (142, 110), (192, 243), (459, 401). De las pruebas hechas con ROS, se observó que los detectores con 3 capas YOLO ayudan a detectar escaleras que están lejos del robot. En estos casos las escaleras aparecen más pequeñas dentro de las imágenes y, por lo tanto, existen otros objetos dentro de las imágenes. Considerando el desempeño de los detectores sobre el robot diferencial y tomando las pruebas hechas con ROS, se decidió utilizar el detector 9 para continuar el desarrollo del *pipeline*.

Tabla 3.2: Configuraciones del modelo YOLOv4-tiny y mAP@0.5IoU obtenidos.

No.	**Iteraciones Totales**	**Subdivisiones**	**Capas totales**	**Capas de detección**	**Parámetro *random***	**Imágenes negativas**	**mAP@0.5**
1	2000	24	38	2	1	NO	59.25 %
2 [4]	2000	24	127	3	0	NO	76.74 %
3 [5]	4000	24	31	3	1	NO	45.24 %
4	4000	24	38	2	0	NO	66.20 %
5	2000	24	38	2	0	NO	63.08 %
6	4000	24	38	2	0	SI	65.53 %
7	10000	8	38	2	0	SI	70.93 %
8	10000	4	45	3	0	SI	68.17 %
9	10000	8	45	3	0	SI	65.32 %

[4] YOLOv4
[5] YOLOv3-tiny

Tabla 3.3: Resultados sobre el conjunto de prueba de cada configuración.

No.	Tiempo de entrena-miento [min]	Tiempo de inferencia aprox. [ms]	Precisión	Recall	F1 score	IoU promedio	mAP@0.5
1	40	5.3	0.76	0.49	0.59	55.09 %	59.25 %
2	186	44.1	0.80	0.69	0.74	63.62 %	76.74 %
3	104	5.6	0.85	0.17	0.29	61.86 %	45.24 %
4	100	5.2	0.73	0.62	0.67	52.76 %	66.20 %
5	60	5.2	0.74	0.56	0.64	53.62 %	63.08 %
6	75	5.47	0.73	0.62	0.67	54.00 %	65.53 %
7	175	5.4	0.81	0.62	0.70	61.36 %	70.93 %
8	160	6.1	0.88	0.52	0.65	67.57 %	68.17 %
9	154	6.2	0.77	0.57	0.65	58.22 %	65.32 %

3.4. Caracterización de la escalera

El detector de escaleras genera una región de interés (ROI por sus siglas en inglés), que permite realizar una caracterización geométrica de las escaleras. Primero, utilizando la caja delimitadora obtenida durante la etapa de detección, se obtiene la nube de puntos correspondiente a esta área. Con esta nube de puntos se realiza la extracción de planos y una caracterización de las escaleras detectadas. El algoritmo de extracción de planos consiste de dos etapas de *clustering*, primero se realiza un algoritmo de crecimiento de regiones para segmentar todas las regiones planas de la nube de puntos de la ROI. Después, se implementa un algoritmo de *clustering* para unir todos los planos con orientaciones y ubicaciones verticales similares. Al hacer esto se obtiene una nube de puntos correspondiente a cada contrahuella y se utiliza para ajustar un plano a la nube de puntos. Con los planos finales se determina la orientación y tamaño de la escalera.

3.4.1. Extracción de planos y caracterización

Utilizando el cuadro delimitador obtenido durante la etapa de detección, se aplica un algoritmo de segmentación y caracterización a la región de interés (ROI) de la nube de puntos para obtener la distancia y la orientación de las escaleras con respecto al sistema de coordenadas de la cámara

(ver Figura 3.8). La ROI inicial, definida por la caja delimitadora, suele tener otros objetos que no corresponden a la huella y contrahuella de la escalera, por ejemplo los pasamanos u objetos del fondo. Para refinar la ROI se redujo el ancho de la caja delimitadora por un factor BB_f que varia proporcionalmente con la distancia hasta la escalera. Esto significa que cuando la distancia es pequeña, el valor de BB_f sera pequeño y vise versa. Los coeficientes de la función lineal que determina el factor de escalamiento se obtuvieron experimentalmente para cada cámara utilizada. Para obtener la ecuación lineal, primero se posicionó al robot a una distancia d_1 del obstáculo y se ajustó el tamaño del cuadro delimitador utilizando un factor BB_{f1} hasta que el ancho del cuadro delimitador fuera igual al ancho del primer escalón. Posteriormente, se posicionó el robot a una distancia $d_2 < d_1$ y se obtuvo el factor BB_{f2} utilizando el mismo procedimiento. Finalmente, se obtuvo la ecuación de la recta que pasa por los puntos (d_1, BB_{f1}) y (d_2, BB_{f2}). Con esta ecuación se pueden determinar el factor de escala BB_f a diferentes distancias de la escalera. Este procedimiento es simple y, utilizando regresión lineal, se pueden utilizar más de dos puntos para mejorar la estimación del factor.

Para extraer los planos de la huella y contrahuella se inicia aplicando un algoritmo de crecimiento de regiones [76]. Este algoritmo consiste de dos etapas principales, la estimación de normales y el crecimiento de regiones. Para calcular las normales de cada punto se ajusta un plano, utilizando mínimos cuadrados, a través de sus k vecinos más cercanos y determina los parámetros de la normal del plano. Al utilizar mínimos cuadrados es posible determinar áreas de gran curvatura utilizando el residuo obtenido durante el ajuste. Posteriormente, la etapa de crecimiento de regiones utiliza las normales y el residuo, en conjunto con restricciones de conectividad local y de suavidad de superficies, para agrupar puntos que pertenecen a la misma región. Finalmente, para cada región, se calcula el centroide y se busca el plano de mejor ajuste utilizando el algoritmo RANSAC [77]. El resultado de estos pasos son los coeficientes, correspondientes a la ecuación $ax + by + cz + d = 0$, de cada región y la ubicación de cada centroide con respecto al sistema coordenada de la cámara.

Utilizando los resultados de las etapas anteriores, se aplica un algoritmo de clustering, que toma en cuenta tres parámetros:

- El ángulo entre cada uno de los planos que fue ajustado a cada región
- La distancia vertical entre cada centroide de cada región
- La distancia lateral entre cada centroide de cada región

El algoritmo de agrupamiento de regiones planas (Algoritmo 1) utiliza los centroides de las regiones planas (todos conforman la nube de puntos $\{P_c\}$), los coeficientes de la normal de cada plano $\{N\}$, los conjuntos de puntos correspondientes a cada región plana $\{C_p\}$ y el número total de direcciones D_{total}, a partir de la semilla inicial, en las cuales buscará. Por otra parte, el algoritmo genera un conjunto de *clusters* $\{C_r\}$que corresponden a cada huella o contrahuella. El algoritmo inicia utilizando una función Ω para encontrar los k vecinos más cercanos de un punto de referencia $(0,0,0)$, considerando solamente los puntos actuales en $\{P_c\}$. Posteriormente, se hace una iteración sobre los k vecinos y se calcula el ángulo entre el plano que representa un vecino y la semilla actual p_s, y la distancia vertical y lateral entre los centroides. Si el ángulo es menor que el umbral Θ_{th} y la distancias son menores que los umbrales d_{vt} (para la distancia vertical) y d_{lt} (para la distancia horizontal), entonces las dos regiones pertenecen a la misma contrahuella. Después de unir estas dos regiones, ahora se utiliza el vecino que cumple los criterios como semilla actual y se elimina de la nube de puntos $\{P_c\}$. El proceso mencionado se repite hasta que no haya puntos que cumplan los criterios y, cuando eso suceda, se utiliza una vez más la semilla inicial p_0 y se vuelven a buscar y unir regiones planas vecinas. Al regresar al punto incial se logra buscar vecinos en D_{total} direcciones a partir de la semilla inicial. El proceso completo continua hasta que todos los puntos en P_c hayan sido semillas.

Una vez que se hayan unido los puntos correspondientes a cada contrahuella, se utiliza el algoritmo RANSAC para encontrar el plano de mejor ajuste para cada *cluster*. Después, para cada plano se determina la orientación del mismo con respecto al plano frontal de la cámara. Para determinar esta orientación es necesario realizar la proyección de la normal del plano correspondiente sobre el plano $y = 0$ (plano del piso). La proyección ortogonal de un vector $\vec{v}$ sobre subespacio H de $\mathbb{R}^n$

Algoritmo 1: Algoritmo para agrupar regiones planas

Entradas: Nube de puntos de centroides = $\{P_c\}$, Normales de cada región = $\{N\}$, Conjuntos de puntos de cada región = $\{C_p\}$, Función para encontrar a los vecinos $\Omega(.)$, Número de direcciones de la búsqueda, a partir de la semilla inicial D_{total}
Salida: Conjuntos de puntos correspondientes a cada huella o contrahuella $\{C_r\}$

```
r ← 0
mientras tamaño({P_c}) > 1 hacer
    Índices de los vecinos más cercanos del punto de referencia {K} ← Ω((0,0,0)) ;
    semilla actual p_s ← K{0} ;
    semilla inicial p_0 ← p_s ;
    P_c —eliminar→ p_0 ;      /* Eliminar semilla de la nube de puntos inicial */
    Índices de los vecinos más cercanos de la semilla actual {K} ← Ω(p_s) ;
    i ← 0;
    d ← 1;
    mientras i < tamaño ({K}) hacer
        θ ← ángulo entre N{p_s} y N{K{i}} ;
        d_z ← distancia entre p_s y K{i} en la dirección z ;
        d_y ← distancia entre p_s y K{i} en la dirección y;
        si | d_z |< d_ht and | d_y |< d_vt and 0 ≤ θ ≤ θ_th entonces
            C_actual ←agregar— puntos de C_p{p_s} C_r ←agregar— puntos de C_current ;
            semilla actual p_s ← K{i} ;
            P_c —eliminar→ p_s ;      /* Eliminar semilla de la nube de puntos */
            Índices de los vecinos más cercanos del punto de referencia {K} ← Ω(p_s) ;
            i ← 0;
        de lo contrario
            i ← i+1 ;
            si i = tamaño ({K}) y d < D_total entonces
                {K} ← Ω(p_0);
                d ← d+1 ;
                i ← 0 ;
            fin
        fin
    fin
    si tamaño (C_r r) >0 entonces
        r ← r+1
    fin
fin
```

con base ortonormal $\{\vec{u}_1, \vec{u}_2, \ldots, \vec{u}_k\}$ esta dada por la ecuación 3.1 [78].

$$proy_H v = (\vec{v} \cdot \vec{u}_1)\,\vec{u}_1 + (\vec{v} \cdot \vec{u}_2)\,\vec{u}_2 + \cdots + (\vec{v} \cdot \vec{u}_k)\,\vec{u}_k \tag{3.1}$$

En el caso del plano $y = 0$, la base ortonormal esta dada por $\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ y $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, y la proyección de cualquier normal $\vec{v}$ sobre el plano $\pi = \left\{ \begin{pmatrix} x \\ y \\ z \end{pmatrix} : y = 0 \right\}$ esta dada por:

$$proy_{\pi}\vec{v} = \left[\vec{v} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \right] \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \left[\vec{v} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right] \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{3.2}$$

Finalmente, el ángulo θ entre la normal $\vec{v}$ y el vector unitario $\vec{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ (el cual representa la dirección lateral en el sistema de coordenadas de la cámara) está dado por:

$$\theta = atan2\left((\vec{x} \times proy_{\pi}\vec{v}) \cdot \vec{y}, \vec{x} \cdot proy_{\pi}\vec{v}\right) \tag{3.3}$$

donde $atan2$ calcula el tangente inverso en los cuatro cuadrantes y $\vec{y} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ es el vector perpendicular a $\vec{x}$ y $proy_{\pi}\vec{v}$. Utilizando la orientación de cada plano se determina si existen por lo menos dos planos cuyas orientaciones son aproximadamente paralelas. Al hacer esto se hace un promedio de las orientaciones para determinar la orientación de la escalera y se utilizan las distancias verticales y laterales de los centroides de los *clusters*, con los cuales se obtuvieron los planos, para determinar el tamaño de la huella y contrahuella.

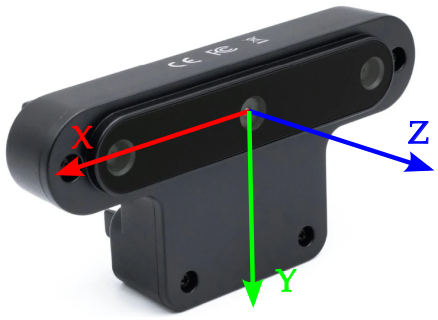


Figura 3.8. Sistema de coordenadas utilizado para la cámara OAK-D.

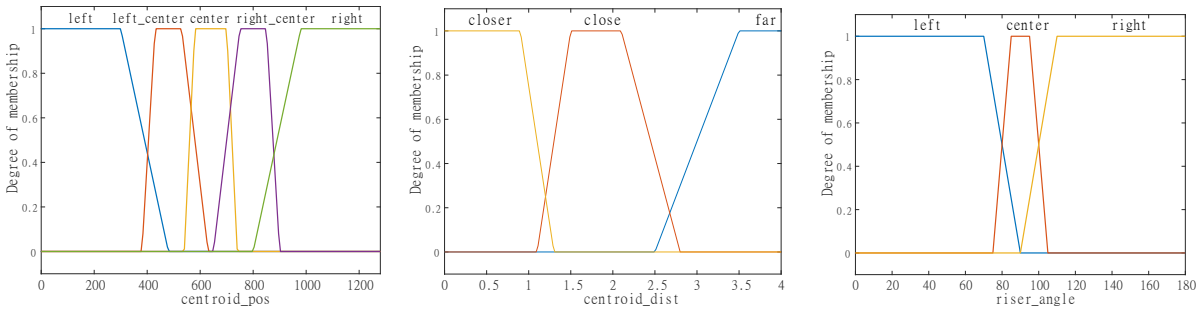
3.5. Desarrollo del sistema de alineación

Para implementar el sistema de alineación se utilizó un controlador difuso que consiste de 45 reglas (ver tabla 3.4) difusas que utilizan 3 entradas: la posición de las escaleras dentro de la imagen a color, la distancia a la escalera y la orientación de las contrahuellas con respecto al robot. El controlador se diseñó de tal manera que durante la primer etapa se utiliza la posición de las escaleras dentro de la imagen a color y la distancia a la escalera para acercarse a ellas hasta que se pueda determinar la orientación de las mismas. Después, el controlador inicia la segunda etapa que se encarga de mover al robot hacia las escaleras de forma paralela a las contrahuellas, pero siempre manteniendo al obstáculo dentro de la imagen RGB. Una vez que el robot se haya acercado lo suficiente a las escaleras, el controlador difuso entra en una tercera fase donde utiliza la orientacion de la misma para alinearse con ellas, de tal manera que este listo para subirlas. Las variables y los conjuntos difusos son los siguientes:

- Variables y conjuntos difusos:
  - Variables de entrada:
    - Coordenada en *x* del centro del cuadro delimitador *(centroid_pos)*.
      - Conjuntos difusos: *left*, *left_center*, *center*, *right_center* y *right*.
    - Distancia hasta el centroide de la nube de puntos de la escalera *(centroid_dist)*.
      - Conjuntos difusos: *closer*, *close* y *far*.

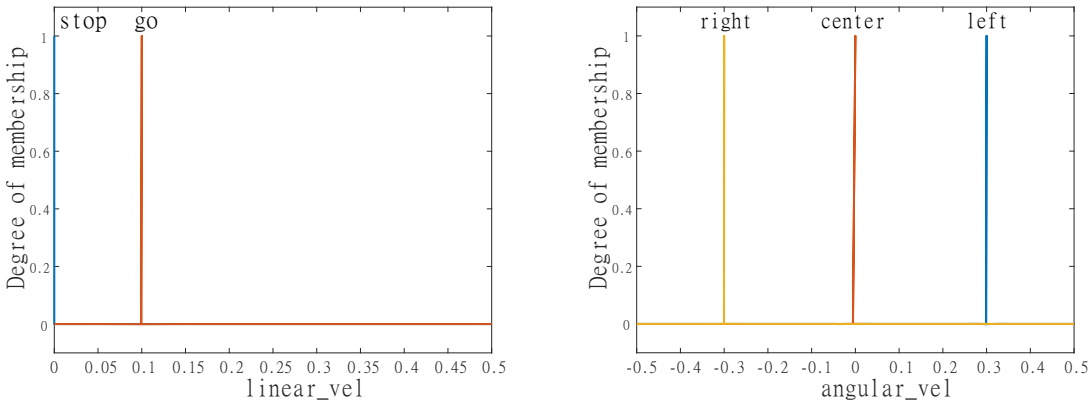
- Staircase orientation *(riser_angle)*.
  - Conjuntos difusos: *left*, *center* y *right*.

- Variables de salida:
  - Velocidad lineal (*linear_vel*).
    - Conjuntos difusos: *stop* y *go*.
  - Velocidad angular (*angular_vel*).
    - Conjuntos difusos: *left*, *center* y *right*.



(a) **(b)** **(c)**

Figura 3.9. Funciones de membresia para las variables de entrada: (**a**) coordenada en x del centro del cuadro delimitador, (**b**) distancia al centroide de la escalera, (**c**) orientación de la escalera con respecto al sistema de coordenadas de la cámara.



(a) **(b)**

Figura 3.10. Funciones de membresia para las variables de salida: (**a**) velocidad lineal, (**b**) velocidad angular.

Las funciones de membresia de los términos lingüísticos se muestran en las Figuras 3.9 y 3.10. La Figura 3.9a muestra las funciones de membresia utilizadas para la variable de entrada *centroid_pos*. El universo de discurso para esta variable es el ancho de la imagen RGB y cada conjunto difuso representa una zona de la imagen. Por otra parte, en la Figura 3.9b se muestran los conjuntos correspondientes a las tres distancias consideradas para el controlador. Para esta variable, el universo de discurso está entre 0 y 4 metros debido a que es el rango de operación de la cámara OAK-D. Cuando el robot se encuentra lejos de la escalera (*far* en inglés), es imposible poder determinar la orientación de las escaleras debido a la poca resolución en la nube de puntos. Una vez que el robot se encuentre cerca o muy cerca (*close* y *closer*, respectivamente) de las escaleras, entonces la nube de puntos mejora los suficiente para poder determinar la orientación de la escalera. Finalmente, en la Figura 3.9c se muestran los conjuntos difusos que representan las tres orientaciones consideradas para el controlador. Debido a el método que se obtuvo para determinar la orientación de las escaleras, el universo de discurso para la variable *riser_angle* está entre 0 y 180 grados. Los valores menores a 90 grados representan una escalera con rotación hacia la izquierda, los valores mayores a 90 grados representan una rotación hacia la derecha y los valores iguales a 90 grados representan una escalera completamente paralela a el plano frontal de la cámara.

En la Figura 3.10 se muestran las funciones de membresia tipo *singleton* utilizados para las variables de salida. Primero, para la variable de velocidad lineal, el universo de discurso está entre 0 y 0.5 y se utilizan los valores de 0 y 0.1 para la velocidad cero (*stop*) y la velocidad lenta (*go*), respectivamente. Por otra parte, para la velocidad angular, el universo de discurso está entre -0.5 y 0.5. Los valores menores a cero representan una rotación a la derecha y los valores mayores a cero una rotación a la izquierda.

3.6. Implementación del pipeline

En la Figura 3.11 se muestran las etapas del sistema o *pipeline* de detección y caracterización de escaleras. Para implementar el pipeline se utilizó el *middleware* conocido como Robot Operating

Tabla 3.4: Reglas difusas utilizadas para el acercamiento y la alineación del robot con las escaleras.

Regla	*IF*			*THEN*	
	centroid_pos AND	centroid_dist AND	riser_angle	linear_vel AND	angular_vel
1	left	far	left	fast	left
2	left	far	center	fast	left
3	left	far	right	fast	left
4	left	close	left	slow	left
5	left	close	center	slow	left
6	left	close	right	slow	left
7	left	closer	left	stop	left
8	left	closer	center	stop	left
9	left	closer	right	stop	left
10	left_center	far	left	fast	left
11	left_center	far	center	fast	left
12	left_center	far	right	fast	left
13	left_center	close	left	slow	right
14	left_center	close	center	slow	left
15	left_center	close	right	slow	left
16	left_center	closer	left	stop	left
17	left_center	closer	center	stop	left
18	left_center	closer	right	stop	left
19	center	far	left	fast	center
20	center	far	center	fast	center
21	center	far	right	fast	center
22	center	close	left	slow	right
23	center	close	center	slow	center
24	center	close	right	slow	left
25	center	closer	left	stop	left
26	center	closer	center	stop	center
27	center	closer	right	stop	right
28	right_center	far	left	fast	right
29	right_center	far	center	fast	right
30	right_center	far	right	fast	right
31	right_center	close	left	slow	right
32	right_center	close	center	slow	right
33	right_center	close	right	slow	left
34	right_center	closer	left	stop	right
35	right_center	closer	center	stop	right
36	right_center	closer	right	stop	right
37	right	far	left	fast	right
38	right	far	center	fast	right
39	right	far	right	fast	right
40	right	close	left	slow	right
41	right	close	center	slow	right
42	right	close	right	slow	right
43	right	closer	left	stop	right
44	right	closer	center	stop	right
45	right	closer	right	stop	right

System o ROS por sus siglas en Ingés. Dos *pipelines* similares se implementaron utilizando dos cámaras RGB-D de bajo costo, la cámara RealSense™ D435i de Intel® y la cámara OpenCV AI Kit with Depth (OAK-D) de Luxonis Holding Corporation. A continuación se describen las cámaras y los nodos que conforman cada *pipeline* implementado en ROS.

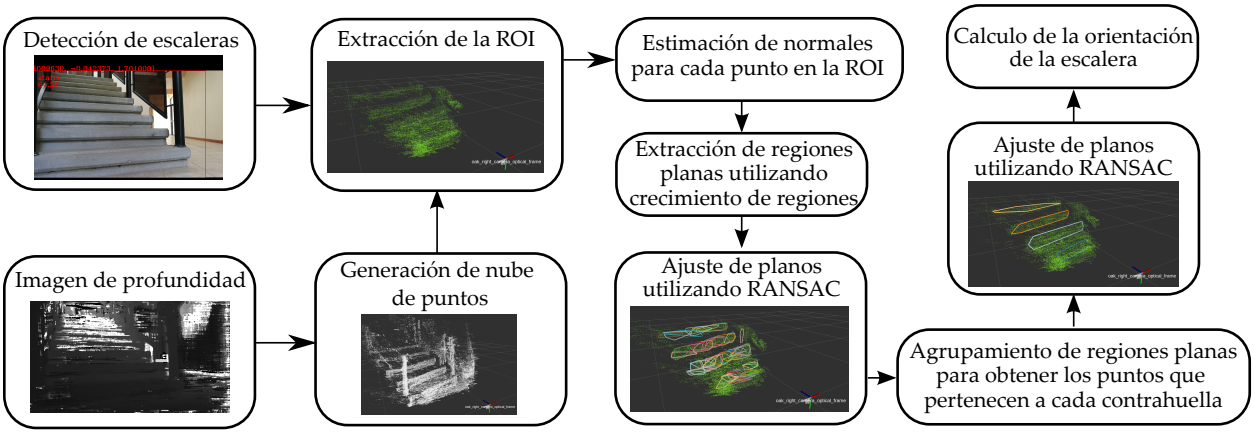


Figura 3.11. Etapas del *pipeline* de detección y caracterización de escaleras.

3.6.1. Descripción de las cámaras

La cámara RealSense™ D435i (Figura 3.12a) es una cámara RGB-D, la cual contiene un sistema de cámaras estéreo, cámara RGB y una unidad de procesador de visión (VPU por sus siglas en Inglés) RealSense D4 Vision de Intel™ . La cámara RGB puede entregar vídeo con resolución de hasta 1920x1080 a 30 cuadros por segundo o FPS. Por otra parte, el sistema estéreo utiliza un proyector de patrones y el VPU para generar imágenes de profundidad con una resolución de hasta 1280x720 y, dependiendo de la resolución, puede reproducirlas a 90 FPS.

Por otra parte, la cámara OAK-D (Figura 3.12b) es una cámara RGB-D que incluye un VPU programable con dos Neural Compute Engines (NCEs) capaces de 1.4 Tera Operaciones por Segundo o TOPs. La habilidad de realizar inferencia neuronal permite implementar redes neuronales en robots con computadoras sin GPU o con pocos recursos computacionales. Esta cámara también es capaz de entregar imágenes a color con una resolución de hasta 4K a 30 FPS e imágenes de profundidad de 1280x720 a 30 FPS.

(a) **(b)**

Figura 3.12. Cámaras utilizadas para los dos *pipelines* implementados en ROS: (**a**) RealSense™ D435i de Intel® [79], (**b**) OAK-D de Luxonis Holding Corporation [80].

3.6.2. Pipelines implementados en ROS

Dos pipelines diferentes se implementaron en ROS utilizando las cámaras descritas anteriormente. Las diferencias principales existen en los nodos de adquisición de imágenes y de detección. Las Figuras 3.13 y 3.14 muestran, de manera simplificada, en donde se ejecutan los nodos para la adquisición de datos y la detección de objetos. Para la cámara D435i se utiliza un *pipeline* que necesita acceso a una GPU, en la computadora del teleoperador, para realizar las detecciones. En comparación, los nodos del *pipeline* para la cámara OAK-D se ejecutan sobre la Computadora a Bordo (OBC por sus siglas en Inglés) y solamente se comunica con la computadora del teleoperador durante el inicio y/o monitoreo de los nodos. Además, debido a que la RealSense tiene más niveles de disparidad, no es necesario realizar el agrupamiento de superficies planas en el *pipeline* de la cámara.

El *pipeline* para la cámara RealSense™ (Figura 3.15) utiliza el *wraper* realsense2_camera [81] para obtener imágenes RGB con una resolución de 848x480 a 30 FPS y publicarlas en el tópico *camera/color/image_raw*. La nube de puntos organizada se obtiene utilizando la imagen de profundidad que genera la cámara y los *nodelets* del paquete *depth_img _proc*. La nube de puntos organizada tiene una dimensión de 848x480 y se publica a una frecuencia de 30 Hz en el tópico

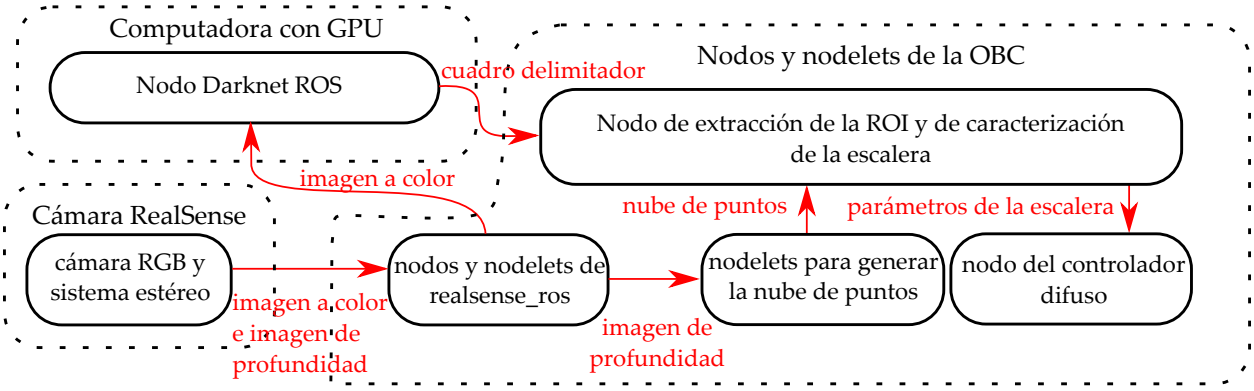


Figura 3.13. Pipeline de ROS simplificado para la cámara RealSense™ D435i. La figura muestra donde se ejecuta cada nodo.

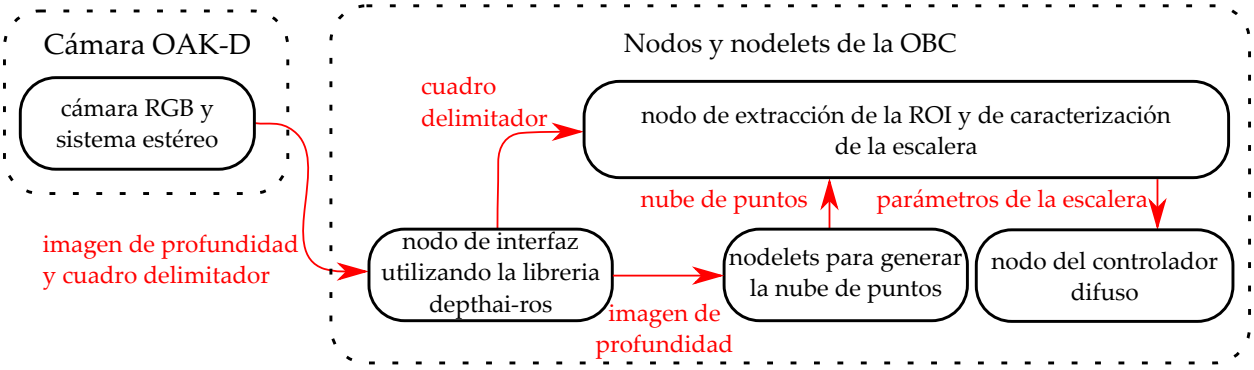


Figura 3.14. Pipeline de ROS simplificado para la cámara OAK-D. La figura muestra donde se ejecuta cada nodo.

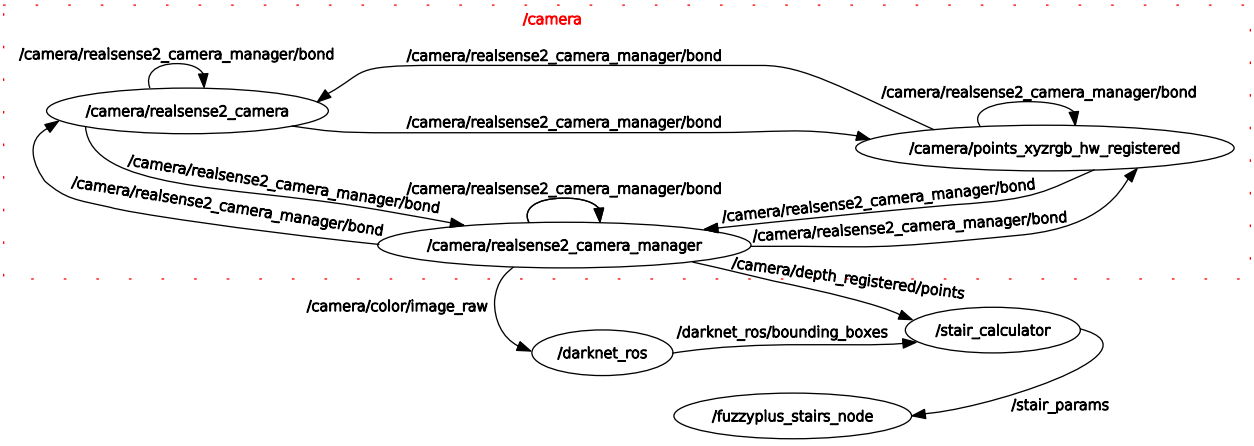


Figura 3.15. Pipeline de ROS para la cámara RealSense™ D435i.

camera/depth_registered/points. Los *nodelets* y sus señales de control (*bonds*) se agrupan en el *namespace /camera* que se muestra en la Figura 3.15.

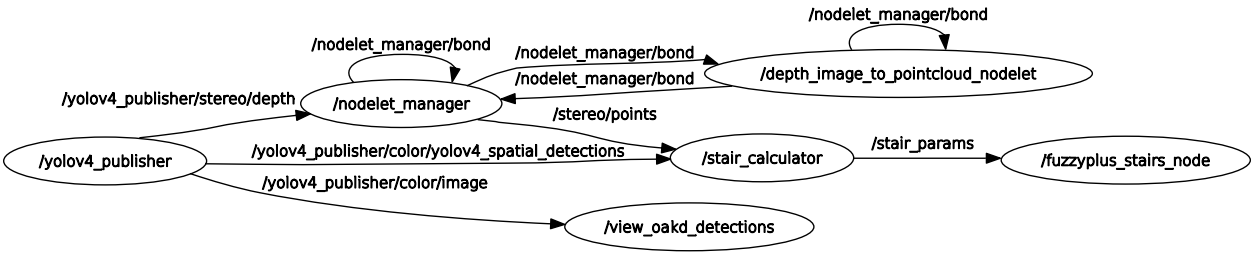


Figura 3.16. Pipeline de ROS para la cámara OAK-D.

El nodo */darknet_ros* realiza las detecciones de escaleras utilizando la imagen RGB obtenida del tópico *camera/color/image_raw* y las publica en el tópico */darknet_ros/bounding_boxes*. Este nodo utiliza el paquete darknet_ros [82], el cual se basa en el *framework* Darknet [75]. Posteriormente, en el nodo */stair_calculator* se encarga de sincronizar la detección y la nube de puntos (obtenida del nodo */camera/realsense2_camera_manager*) para realizar el cálculo de los parámetros de la escalera y publicarlos en el topico */stair_params*. Finalmente, el nodo */fuzzyplus_stairs_node* se subscribe al tópico */stair_params* y realiza la inferencia del sistema difuso y publica los comandos de velocidad necesarios para mover al robot.

Antes de implementar el *pipeline* de la cámara OAK-D (Figura 3.16), fue necesaria una conversión de los pesos de la red neuronal utilizada por Darknet a un formato *.blob* requerido para ser ejecutado en el hardware de la OAK-D (ver Figura 3.17). El primer paso consiste en convertir los pesos del modelo utilizado en Darknet a un modelo congelado de TensorFlow (*TensorFlow frozen model*) utilizando el programa en Python *convert_weights_pb.py* del repositorio de TNTWEN [83]. Posteriormente, utilizando el Optimizador de Modelos del Toolkit de OpenVINO™ [84], se hace la conversión del modelo congelado de TensorFlow a Intermediate Representation (IR), obteniendo así un archivo *.xml* con la configuración de la red neuronal y un archivo *.bin* que contiene los pesos del modelo entrenado. Finalmente, utilizando los archivos obtenidos con el Optimizador de Modelos, se utiliza el Compilador de OpenVINO™ para generar el archivo *.blob* requerido por el VPU de la cámara OAK-D.

Para obtener las imágenes y detecciones que realiza la cámara OAK-D se utilizó el paquete depthai-ros proporcionado por Luxonis [86]. Utilizando este paquete se programó el nodo */yo-*

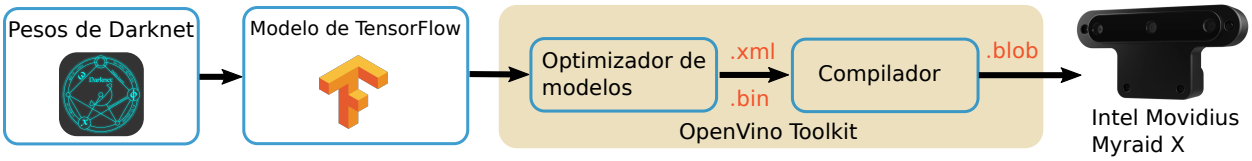


Figura 3.17. Pasos necesarios para convertir los pesos de Darknet al formato .blob. [85].

lov4_publisher (ver Figura 3.16), el cual, realiza la configuración de la red neuronal correspondiente al modelo en formato *.blob* y configura la cámara de tal manera que la detección se realice utilizando imágenes RGB con una resolución de 1920x1080 y a 30 FPS. De igual manera, en este nodo se configuraron las cámaras estéreo con una resolución de 1280×720 y a 30 FPS y también se sincronizaron las detecciones de escaleras y las imágenes de profundidad. Debido a que las resoluciones de la cámara RGB y las cámaras del sistema estéreo son diferentes, el VPU se encarga de realizar una alineación de las imágenes. Después de las configuraciones de la cámara, el nodo publica la imagen de profundidad, las detecciones obtenidas y la imagen RGB en los tópicos */yolov4_publisher/stereo/depth*, */yolov4_publisher/color/yolov4_spatial_detections* y */yolov4_publisher/color/image*, respectivamente, a una frecuencia de 18 Hz. Finalmente, para calcular los parámetros de la escalera y para el controlador difuso se utilizaron los nodos */stair_calculator* y */fuzzyplus_stairs_node*, respectivamente.

Capítulo 4

Resultados experimentales

En este capítulo se presentan las detecciones obtenidas en distintas imágenes del conjunto de prueba utilizando el detector entrenado en Google Colab. Posteriormente, se muestra la detección y caracterización de las escaleras utilizando los dos *pipelines* implementados en ROS. Además, utilizando el controlador difuso desarrollado, se realizan pruebas de acercamiento y alineación. Cabe mencionar que los resultados obtenidos de este trabajo se publicaron en la revista científica MDPI Applied Sciences en el artículo titulado "Staircase Detection, Characterization and Approach Pipeline for Search and Rescue Robots" [87].

4.1. Resultados de detección en imágenes de prueba

Las evaluaciones de las tablas 3.2 y 3.3 se obtuvieron utilizando el conjunto de imágenes de prueba y a continuación se muestran ejemplos de algunas detecciones obtenidas. En las Figuras 4.1, 4.2 y 4.3 se observan las predicciones que realiza el detector sobre 6 imágenes tomadas del conjunto de prueba. En las primeras 2 imágenes se observa que el modelo detecta escaleras con distintas texturas y en imágenes tomadas con diferentes puntos de vista. Además, en comparación con otros métodos descritos en el estado del arte, este modelo logra detectar escaleras de caracol y escaleras donde las contrahuellas no están presentes físicamente. En la Figura 4.2 se detectan

Figura 4.1. Predicciones obtenidas utilizando imágenes del conjunto de prueba pertenecientes a la base de datos Open Images V6.

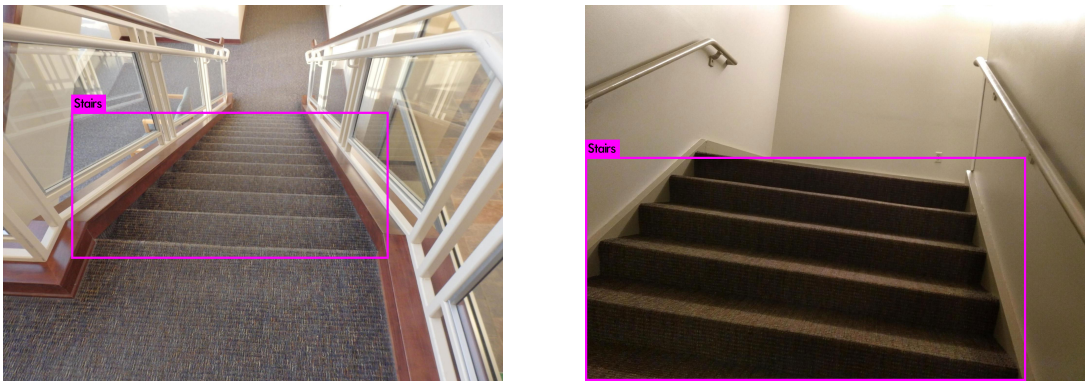


Figura 4.2. Predicciones obtenidas utilizando imágenes del conjunto de prueba pertenecientes a la base de datos MCIndoor20000.

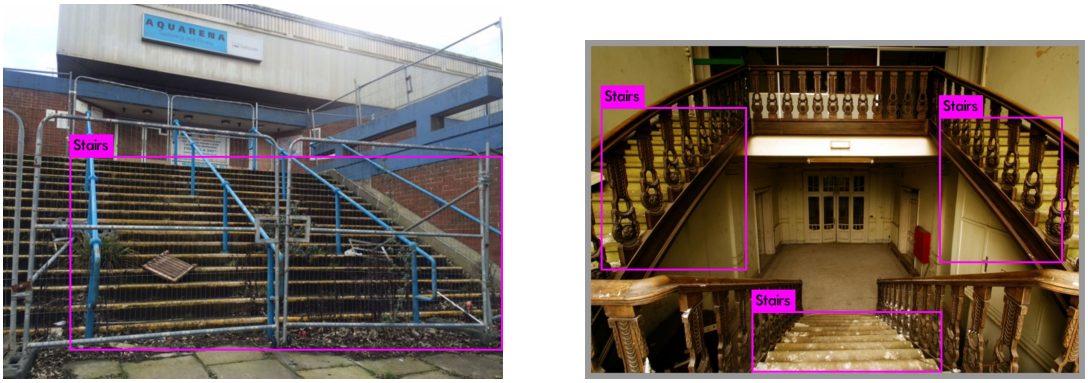
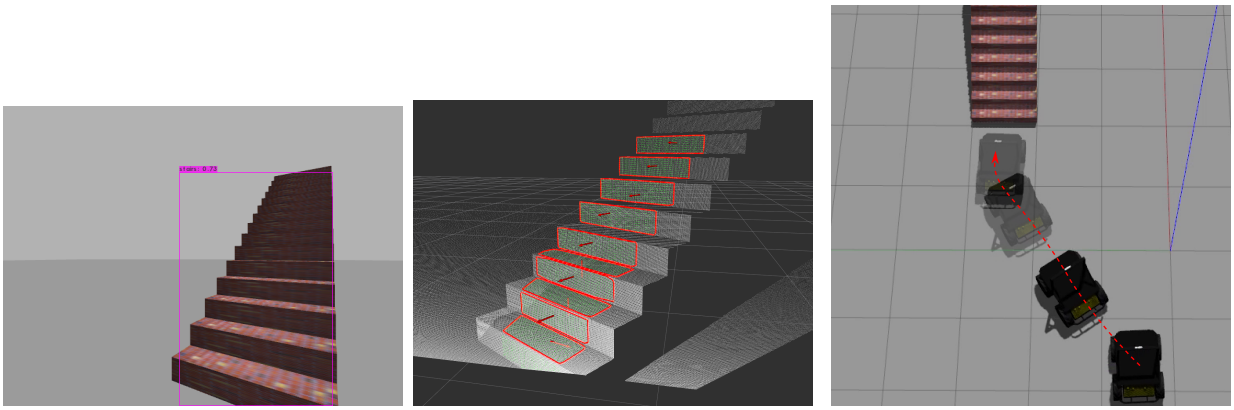


Figura 4.3. Predicciones obtenidas utilizando imágenes del conjunto de prueba pertenecientes a la base de datos Open Images V6.

escaleras ascendentes y descendentes y en ubicaciones con distintos niveles de iluminación. Finalmente, en la Figura 4.3 se observa que el modelo detecta múltiples escaleras en una imagen y también escaleras que están parcialmente ocluidas por cercas o pasamanos.



(a) (b) (c)

Figura 4.4. Prueba del *pipeline* en simulación: (**a**) detección, (**b**) segmentación de regiones planas, (**c**) acercamiento y alineación utilizando el controlador difuso.

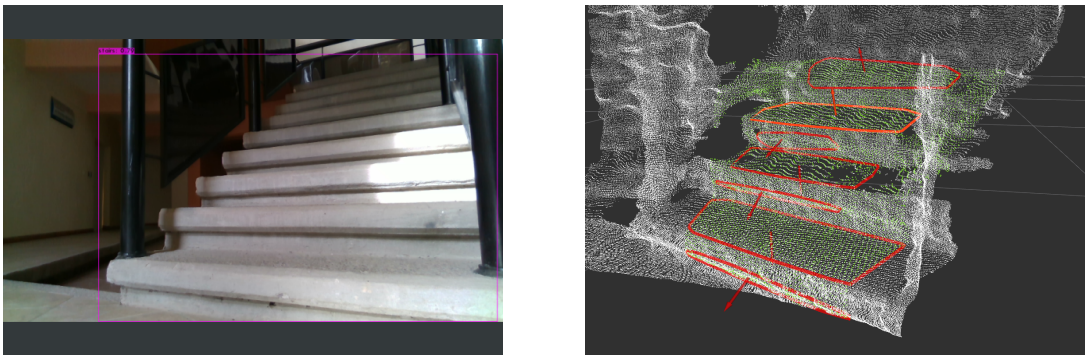
4.2. Implementación de los *pipelines*

Antes de implementar los *pipelines* sobre el robot diferencial, se decidió implementarlo utilizando una simulación del robot Husky de Clearpath Robotics [88]. La simulación del *pipeline* permitió probar diferentes reglas difusas y ajustar manualmente los parámetros del controlador difuso para obtener así valores iniciales para ser utilizados con el robot diferencial real. Debido a que el paquete del robot Husky utiliza a la cámara RealSense durante la simulación, se utilizó el pipeline correspondiente. Además, la nube de puntos generada es ideal y, por lo tanto, solo se necesita implementar el algoritmo de crecimiento de regiones para obtener los puntos correspondientes a las huellas y contrahuellas. En la Figura 4.4 se muestra la detección y caracterización de la escalera y, además, el camino que el robot toma para alinearse con ellas.

Para implementar los *pipelines* utilizando las cámaras reales se utilizaron dos computadoras, una como OBC y la otra para el teleoperador. Las especificaciones de cada máquina se muestran en la Tabla 4.1. Para ambos *pipelines* la cámara se montó sobre el robot diferencial y se posicionó en frente de las escaleras. Los resultados obtenidos utilizando la cámara D435i se muestran en la Figura 4.5. En la Figura 4.5a se observa la detección de la escalera en la imagen RGB que se utiliza para la extracción de los puntos de la ROI que se muestran de color verde en la Figura 4.5b. Debido a la calidad la nube de puntos de esta cámara, en comparación con la cámara OAK-D,

Tabla 4.1: Especificaciones de las máquinas utilizadas para implementar el *pipeline*.

Característica	**Especificación**	
	Computadora a bordo	Computadora del teleoperador
CPU	Intel® Core™ i5-2450M CPU @ 2.50GHz	Intel® Core™ i7-10750H CPU @ 2.60GHz
GPU	NA	Nvidia® RTX 2070 con diseño Max-Q
Sistema Operativo	Ubuntu 18.04.1 LTS bionic	Ubuntu 18.04.3 LTS bionic
Middleware	ROS Melodic	ROS Melodic
Versión de *driver* de Nvidia®	NA	460.39
Versión de CUDA	NA	11.2.2



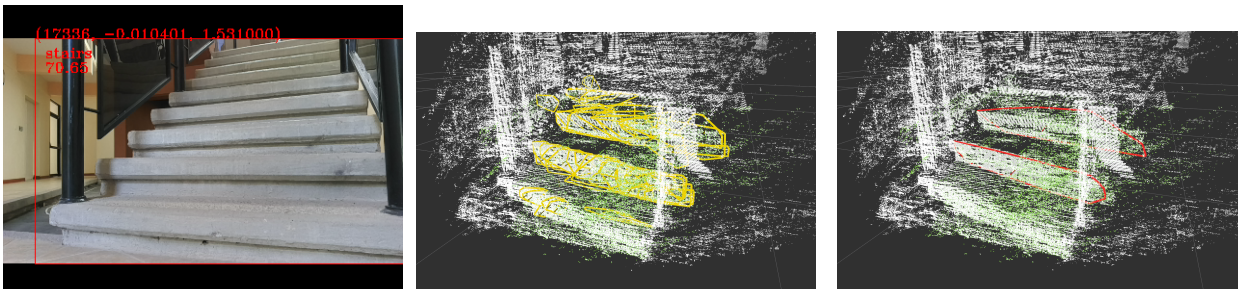
(a) **(b)**

Figura 4.5. Resultados obtenidos utilizando el *pipeline* de la cámara D435i: (**a**) detección, (**b**) segmentación de regiones planas.

la segmentación de las huellas y contrahuellas se realizo utilizando solamente el algoritmo de crecimiento de regiones y los planos detectados se muestran de color rojo en la Figura 4.5b.

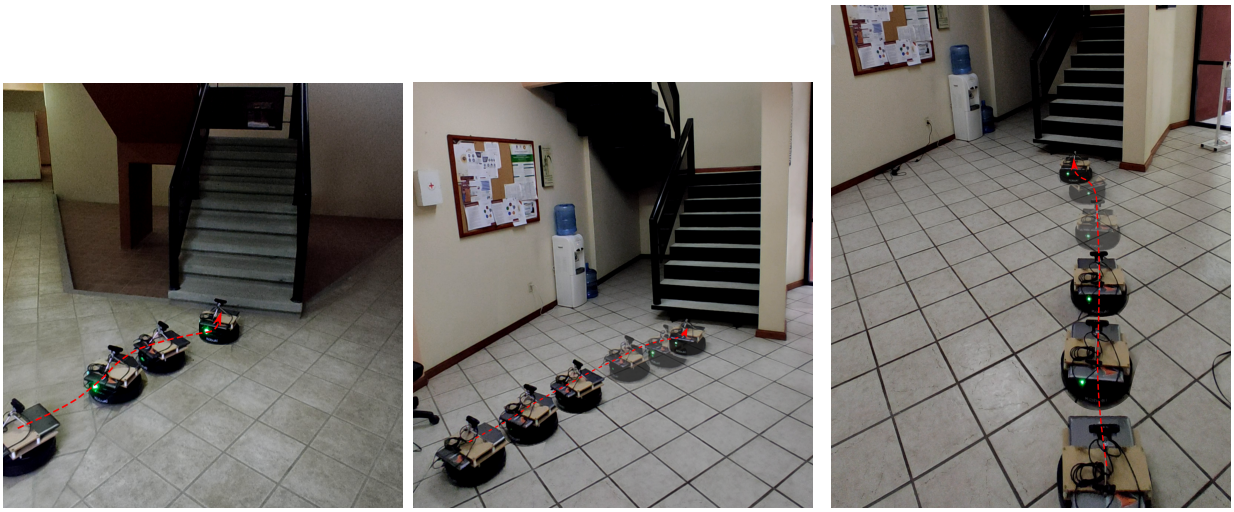
En la Figura 4.6a se muestra la detección obtenida utilizando el *pipeline* de la cámara OAK-D utilizando la imagen RGB y los NCEs cámara. Posteriormente, utilizando el cuadro delimitador, se extraen los puntos correspondientes a la ROI (puntos verdes en las Figuras 4.6b y 4.6c) y se hace la segmentación de regiones planas para ajustar planos a cada una (planos amarillos en la Figura 4.6b). Finalmente, en última etapa se hace el agrupamiento de las regiones planas para obtener las regiones correspondientes a las contrahuellas (planos rojos de la Figura 4.6c).

Considerando todos los componentes necesarios para la detección y caracterización de la escalera, ambos *pipelines* publican los parámetros de las escaleras a 2 Hz. Debido a los bajos recursos



(a) **(b)** **(c)**

Figura 4.6. Resultados obtenidos utilizando el *pipeline* de la cámara OAK-D: (**a**) segmentación de regiones planas, (**b**) agrupamiento de regiones planas.



(a) **(b)** **(c)**

Figura 4.7. Camino que sigue el robot al utilizar el controlador difuso para acercarse y alinearse con las escaleras cuando se posiciona: (**a**) a la izquierda de escalera 1, (**b**) a la izquierda de la escalera 2, (**c**) a la derecha de la escalera 2.

de computo de la OBC del robot diferencial y considerando el uso del *pipeline* para el robot de rescate construido en la universidad, se decidió utilizar la cámara OAK-D para probar el controlador difuso. Este controlador utiliza los parámetros de la escalera para realizar inferencia difusa y publicar los comandos de velocidad lineal y angular necesarios para que el robot se acerque a las escaleras y se alinee con ellas. Para probar el controlador difuso, el robot se posicionó en diferentes posiciones y distancias con respecto a las escaleras. Después, el *pipeline* completo se ejecutó desde la máquina del teleoperador. Al hacer esto, el robot se mueve lentamente hacia las escaleras hasta que logra determinar su orientación (a aproximadamente 1.5 metros de distancia). Con la orienta-

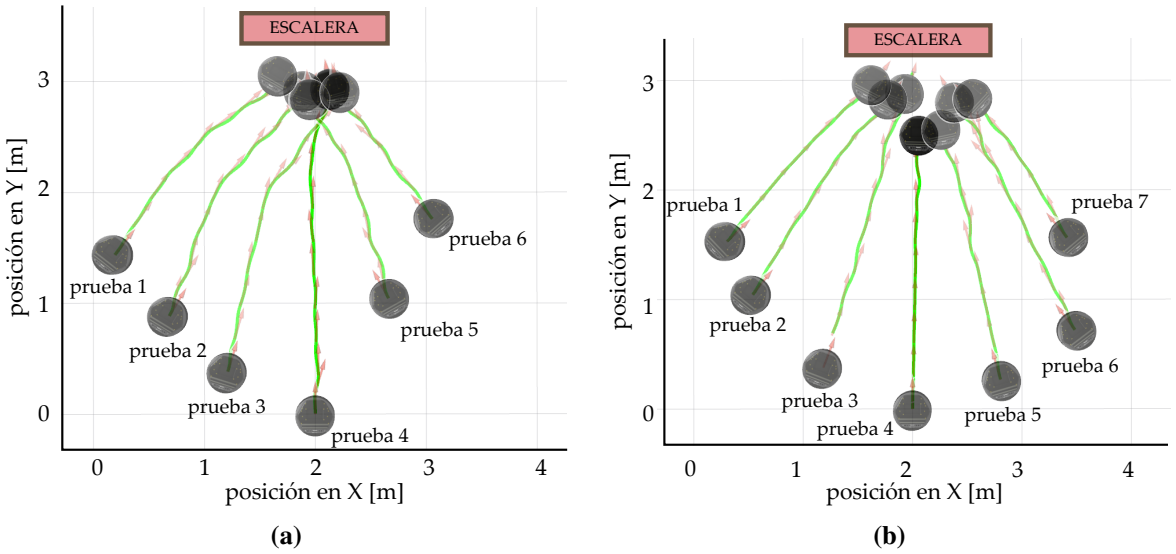


Figura 4.8. Caminos generados por el controlador difusa: (**a**) pruebas hechas con la escalera 1, (**b**) pruebas hechas con las escalera 2.

ción, el robot se mueve de tal manera que se encuentre de frente en el centro de las escaleras. El robot termina de moverse cuando este alineado y a una distancia de aproximadamente 90 cm del centroide de la nube de puntos de la ROI.

En la Figura 4.7 se muestran los caminos que se generan utilizando las escaleras 1 y 2. En cada figura, el camino que toma el robot para llegar hasta la escalera se muestra con una linea punteada de color rojo. Adicionalmente, la Figura 4.8 muestran los caminos obtenidos en 13 pruebas diferentes utilizando las escaleras 1 y 2. A partir de estas gráficas se observó que el *pipeline* tiene dificultades para poder alinearse con escaleras que no tienen textura en su contra huella (por ejemplo la escalera 2). Esto se debe a que el sistema estéreo necesita la textura para poder generar la imagen de profundidad y posteriormente los *nodos* y *nodelets* de ROS necesitan la imagen de profundidad para generar la nube de puntos. En imágenes sin textura o cuando el robot está muy lejos de las escaleras, la nube de puntos tiende a fallar debido a que no se puede realizar el cálculo de las disparidades correctamente y a causa de esto las nubes de puntos presentan mucho ruido y un error grande y, por lo tanto, no se pueden ajustar los planos a las contrahuellas ni determinar su orientación. Las pruebas también muestran que el controlador difuso tiene dificultad para posicionar el robot correctamente cuando se posiciona a mas de 1.5m a la izquierda o derecha del centro

del obstáculo.

Los resultados muestran que la detección y caracterización funciona con diferentes cámaras y que puede ser utilizado para realizar un acercamiento y alineamiento con el obstáculo. Además, al utilizar submuestreo, crecimiento de regiones y algoritmos de *clustering* fue posible reducir los efectos de ruido durante la segmentación de planos y disminuir el tiempo de ejecución. A causa de esto, el algoritmo puede ser utilizado con nubes de puntos generadas de cámaras con bajos niveles de disparidad o con mucho ruido. Por otra parte, la velocidad de ejecución del algoritmo fue lo suficientemente veloz para poder implementar un controlador difuso sobre un robot real.

Capítulo 5

Conclusiones y trabajos futuros

Utilizando aprendizaje profundo fue posible desarrollar e implementar un *pipeline* para la detección y caracterización de escaleras. El *pipeline* propuesto mejora significativamente la teleoperabilidad y autonomía de un robot diferencial al permitir que el robot se alinee, de manera autónoma, con el obstáculo. Utilizando la arquitectura general del *pipeline*, fue posible implementarlo utilizando dos cámaras diferentes. Al hacer esto, se logró validar su funcionamiento y flexibilidad para ser utilizado en robots con diferentes especificaciones y requerimientos.

Para seleccionar y adaptar la arquitectura YOLOv4-tiny al *pipeline* fue necesario considerar distintas métricas de evaluación y tomar en cuenta los requerimientos computacionales del robot de búsqueda y rescate. Además, YOLOv4-tiny fue comparado con las arquitecturas más actuales y se comparó su desempeño al entrenar tres arquitecturas diferentes y obtener las métricas correspondientes. Posteriormente, para optimizar el detector, se hicieron cambios en su arquitectura y se modificaron distintos parámetros de entrenamiento que mejoran su desempeño sobre el conjunto de prueba. Los detectores optimizados fueron implementados directamente con las cámaras y se observó que los detectores con 3 capas de detección tienen un mejor funcionamiento con imágenes de tamaño 1920x1080. Esto presentó una ventaja clara debido a que el robot logra tener un mayor rango de visión sin afectar la velocidad de detección.

Haciendo uso del detector de escaleras y tomando en cuenta los requerimientos para su uso con

sensores ruidosos y de bajo costo, primero se consideró una segmentación de planos utilizando componentes conectados. Sin embargo, se observó que este algoritmo solamente es viable para cámaras con altos niveles de disparidad y poco ruido. A causa de esto se optó por diseñar un algoritmo con mayor robustez. El algoritmo consiste en realizar un submuestreo de la nube de puntos y realizar dos etapas de extracción de planos y de agrupamiento. Al hacer esto, fue posible segmentar las huellas y contrahuellas de las escaleras utilizando la cámara OAK-D la cual solamente utiliza 96 niveles de disparidad comparados con los 128 que la RealSense™ utiliza. Adicionalmente, el algoritmo logra segmentar escaleras en donde la contrahuella no es completamente plana.

Finalmente, el algoritmo de detección y caracterización se utilizó para implementar un controlador difuso que permitió el acercamiento y alineación con el obstáculo. La frecuencia con la que se obtiene la distancia y orientación de la escalera fue suficiente para implementar el controlador difuso. Además, los recursos computacionales necesarios para todo el *pipeline* muestra, una vez más, que es viable para el uso dentro de la robótica para desastres. En comparación, los métodos del estado del arte no consideran el uso de sus propuestas en sistemas limitados o sistemas de bajo costo.

El sistema propuesto tiene ciertas limitaciones, que se mencionan a continuación. Primero, debido a el tipo de tecnología utilizada para obtener la nube de puntos, la etapa de alineación esta limitada a ser utilizada a distancias cortas. Esto significa que aunque las escaleras se lograron detectar a una distancia mayor a 8 metros, la resolución de la nube de puntos a esta distancia es muy baja y es imposible obtener las características del obstáculo. Afortunadamente, en estos casos, el diseño del controlador difuso permitió que el robot se acerque (sin realizar una alineación) al obstáculo aún cuando se desconoce la orientación del obstáculo.

Por otra parte, debido a que los detectores de objetos tienen como salida un cuadro delimitador que enmarca al obstáculo, se genera un problema cuando el obstáculo se observa desde los lados. En estos casos el cuadro delimitador no enmarca perfectamente al obstáculo y pueden existir otros objetos dentro del mismo. En este trabajo, este problema se resolvió utilizando un escalamiento del cuadro delimitador. Sin embargo, no es posible cubrir todos los casos de cuando esto sucede

y por lo tanto otra opción seria realizar una segmentación a nivel de pixeles. De esta manera se podría segmentar exactamente las huellas y contrahuellas antes de realizar la caracterización de las mismas.

Debido a la arquitectura del *pipeline* propuesto, es posible escalarlo para detectar y caracterizar más obstáculos, tales como rampas o escombro, que un robot puede encontrar durante su recorrido. Para un trabajo futuro se ha considerado entrenar el detector con imágenes de otros obstáculos comunes en zonas de desastres. Posteriormente, para cada obstáculo se puede realizar la caracterización necesaria para que facilite la toma de decisiones y/o el cruce del obstáculo.

Considerando las limitaciones del controlador difuso, la mejor manera de implementar un sistema de acercamiento y alineación sería implementando un algoritmo de planeación de trayectorias que utilice la posición y la orientación de las escaleras como entradas. Al hacer esto no sería necesario mantener a las escaleras dentro de la imagen RGB durante el movimiento del robot. Esto sería una mejora considerable y permitiría que el robot se alinee adecuadamente aún cuando se posicione a distancias mayores a 1.5m a la derecha o a la izquierda del centro de la escalera.

Bibliografía

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 9, pp. 1904–1916, 2015.

[2] A. Davids, "Urban search and rescue robots: from tragedy to technology," *IEEE Intelligent systems*, vol. 17, no. 2, pp. 81–83, 2002.

[3] R. R. Murphy, *Disaster robotics*. Intelligent robotics and autonomous agents, MIT press, 2014.

[4] Y. Cong, X. Li, J. Liu, and Y. Tang, "A stairway detection algorithm based on vision for UGV stair climbing," *Proceedings of 2008 IEEE International Conference on Networking, Sensing and Control, ICNSC*, pp. 1806–1811, 2008.

[5] C. Zhong, Y. Zhuang, and W. Wang, "Stairway detection using Gabor filter and FFPG," *Proceedings of the 2011 International Conference of Soft Computing and Pattern Recognition, SoCPaR 2011*, pp. 578–582, 2011.

[6] T. Westfechtel, K. Ohno, B. Mertsching, R. Hamada, D. Nickchen, S. Kojima, and S. Tadokoro, "Robust stairway-detection and localization method for mobile robots using a graph-based model and competing initializations," *International Journal of Robotics Research*, vol. 37, no. 12, pp. 1463–1483, 2018.

[7] T. Westfechtel, K. Ohno, B. Mertsching, D. Nickchen, S. Kojima, and S. Tadokoro, "3D graph based stairway detection and localization for mobile robots," *IEEE International Conference on Intelligent Robots and Systems*, vol. 2016-November, pp. 473–479, 2016.

[8] D. C. Hernández and K.-h. Jo, "Stairway Tracking Based on Automatic Target Selection Using Directional Filters," *17th Korea-Japan Joint Workshop on Frontiers of Computer Vision (FCV)*, pp. 1–6, 2011.

[9] S. Shahrabadi, J. M. Rodrigues, and J. M. Du Buf, "Detection of indoor and outdoor stairs," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7887 LNCS, pp. 847–854, 2013.

[10] J. A. Delmerico, D. Baran, P. David, J. Ryde, and J. J. Corso, "Ascending stairway modeling from dense depth imagery for traversability analysis," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 2283–2290, 2013.

[11] W. Samakming and J. Srinonchat, "Development image processing technique for climbing stair of small humanoid robot," *Proceedings of the International Conference on Computer Science and Information Technology, ICCSIT 2008*, pp. 616–622, 2008.

[12] S. Murakami, M. Shimakawa, K. Kivota, and T. Kato, "Study on stairs detection using RGB-depth images," *2014 Joint 7th International Conference on Soft Computing and Intelligent Systems, SCIS 2014 and 15th International Symposium on Advanced Intelligent Systems, ISIS 2014*, pp. 1186–1191, 2014.

[13] Q. Zhang, S. S. Ge, and P. Y. Tao, "Autonomous stair climbing for mobile tracked robot," *9th IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR 2011*, pp. 92–98, 2011.

[14] S. Oßwald, J. S. Gutmann, A. Hornung, and M. Bennewitz, "From 3D point clouds to climbing stairs: A comparison of plane segmentation approaches for humanoids," *IEEE-RAS International Conference on Humanoid Robots*, pp. 93–98, 2011.

[15] E. Mihankhah, A. Kalantari, E. Aboosaeedan, H. D. Taghirad, and S. A. A. Moosavian, "Autonomous staircase detection and stair climbing for a tracked mobile robot using fuzzy controller," *2008 IEEE International Conference on Robotics and Biomimetics, ROBIO 2008*, pp. 1980–1985, 2008.

[16] A. I. Mourikis, N. Trawny, S. I. Roumeliotis, D. M. Helmick, and L. Matthies, "Autonomous stair climbing for tracked vehicles," *International Journal of Robotics Research*, vol. 26, no. 7, pp. 737–758, 2007.

[17] Y. Okada, K. Nagatani, and K. Yoshida, "Semi-autonomous operation of tracked vehicles on rough terrain using autonomous control of active flippers," *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2009*, pp. 2815–2820, 2009.

[18] J. A. Hesch, G. L. Mariottini, and S. I. Roumeliotis, "Descending-stair detection, approach, and traversal with an autonomous tracked vehicle," *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*, pp. 5525–5531, 2010.

[19] L. A. Souto, A. Castro, L. M. G. Gonçalves, and T. P. Nascimento, "Stairs and doors recognition as natural landmarks based on clouds of 3D edge-points from RGB-D sensors for mobile robot localization," *Sensors (Switzerland)*, vol. 17, no. 8, pp. 1–16, 2017.

[20] A. Sinha, P. Papadakis, and M. R. Elara, "A staircase detection method for 3D point clouds," *2014 13th International Conference on Control Automation Robotics and Vision, ICARCV 2014*, vol. 2014, no. December, pp. 652–656, 2014.

[21] M. Ilyas, A. K. Lakshmanan, A. V. Le, and M. R. Elara, "Staircase Recognition and Localization using Convolution Neural Network (CNN) for Cleaning Robot Application," no. December, pp. 1–16, 2018.

[22] O. K. e. Bruno Siciliano, *Springer Handbook of Robotics*. Springer International Publishing, 2 ed., 2016.

[23] A. Jacoff, E. Messina, and J. Evans, "Experiences in deploying test arenas for autonomous mobile robots," tech. rep., National Institute of Standards and Technology Gaithersburg MD Intelligent Divisions, 2001.

[24] J. A. Sánchez, "Diseño y construcción de un sistema de tracción, suspensión y chasis para un prototipo de robot de búsqueda y rescate," tesis, Universidad Tecnológica de la Mixteca, Huajuapan de León, Oaxaca, 2020.

[25] V. E. S. Luna, "Diseño e implementación de la interfaz electrónica para el robot de búsqueda y rescate msv-01 y su control e integración con ros," tesis, Universidad Tecnológica de la Mixteca, Huajuapan de León, Oaxaca, 2020.

[26] Ubuntu, "Ubuntu." Disponible en: `https://ubuntu.com/`. Accedido: 2020-20-04.

[27] ROS, "About ros." Disponible en: `https://www.ros.org/about-ros/`. Accedido: 2020-20-04.

[28] K. Dawson-Howe, *A practical introduction to computer vision with opencv*. John Wiley & Sons, 2014.

[29] R. C. Gonzalez, R. E. Woods, *et al.*, "Digital image processing," 2002.

[30] R. Szeliski, *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[31] A. De la Escalera, *Visión por computador: fundamentos y métodos*. Pearson, Prentice Hall, 2001.

[32] M. Nixon and A. Aguado, *Feature extraction and image processing for computer vision*. Academic Press, 2019.

[33] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[34] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[35] C. C. Aggarwal, *Neural networks and deep learning*, vol. 10. Springer, 2018.

[36] F. Berzal, *Redes neuronales & deep learning*. Editorial Universidad de Granada, 2018.

[37] A. Deshpande, "A beginner's guide to understanding convolutional neural networks." Disponible en: `https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/`. Accedido: 2019-16-02.

[38] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *32nd International Conference on Machine Learning, ICML 2015*, vol. 1, pp. 448–456, 2015.

[39] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *proc. OF THE IEEE*, 1998.

[40] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[41] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[42] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[43] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection," 2020.

[44] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems*, vol. 28, pp. 91–99, 2015.

[45] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[46] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, pp. 1440–1448, 2015.

[47] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9905 LNCS, pp. 21–37, 2016.

[48] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[49] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[50] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 834–848, 2017.

[51] S. Liu, D. Huang, *et al.*, "Receptive field block net for accurate and fast object detection," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 385–400, 2018.

[52] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[53] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, "Path aggregation network for instance segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8759–8768, 2018.

[54] AlexeyAB, "Alexeyab/darknet." Disponible en : `https://github.com/AlexeyAB/darknet#how-to-improve-object-detection`.

[55] R. Padilla, S. L. Netto, and E. A. da Silva, "A survey on performance metrics for object-detection algorithms," in *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*, pp. 237–242, IEEE, 2020.

[56] J.-S. R. Jang, C.-T. Sun, E. Mizutani, and S. Computing, "A computational approach to learning and machine intelligence," *IEEE Transactions on automatic control*, vol. 42, no. 10, pp. 1482–1484, 1997.

[57] A. T. Velázquez, "Notas de clase en fundamentos de inteligencia artificial," Febrero 2020.

[58] L. A. Zadeh, "Fuzzy sets," in *Fuzzy sets, fuzzy logic, and fuzzy systems: selected papers by Lotfi A Zadeh*, pp. 394–432, World Scientific, 1996.

[59] L. A. Zadeh, "Outline of a new approach to the analysis of complex systems and decision processes," *IEEE Transactions on systems, Man, and Cybernetics*, no. 1, pp. 28–44, 1973.

[60] E. H. Mamdani and S. Assilian, "An experiment in linguistic synthesis with a fuzzy logic controller," *International journal of man-machine studies*, vol. 7, no. 1, pp. 1–13, 1975.

[61] MathWorks, "Fuzzy inference process." Disponible en: `https://www.mathworks.com/help/fuzzy/fuzzy-inference-process.html`, 2021.

[62] F. S. Bashiri, E. LaRose, P. Peissig, and A. P. Tafti, "MCIndoor20000: A fully-labeled image dataset to advance indoor objects detection," *Data in Brief*, vol. 17, pp. 71–75, 2018.

[63] Tzutalin, "Labelimg." Disponible en: `https://github.com/tzutalin/labelImg`, 2015. Accedido: 2020-20-04.

[64] Google, "Google images." Disponible en: `https://www.google.com/imghp?hl=en`. Accedido: 2020-20-11.

[65] A. Rosenbrock, "How to create a deep learning dataset using google images." Disponible en: `https://www.pyimagesearch.com/2017/12/04/how-to-create-a-deep-learning-dataset-using-google-images/`. Accedido: 2020-20-11.

[66] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari, "The Open Images Dataset V4: Unified Image Classification, Object Detection, and Visual Relationship Detection at Scale," *International Journal of Computer Vision*, vol. 128, no. 7, pp. 1956–1981, 2020.

[67] I. Krasin, T. Duerig, N. Alldrin, V. Ferrari, S. Abu-El-Haija, A. Kuznetsova, H. Rom, J. Uijlings, S. Popov, A. Veit, S. Belongie, V. Gomes, A. Gupta, C. Sun, G. Chechik, D. Cai, Z. Feng, D. Narayanan, and K. Murphy, "Openimages: A public dataset for large-scale multi-label and multi-class image classification.," *Dataset available from https://github.com/openimages*, 2017.

[68] Huang, Jonathan and Rathod, Vivek and Sun, Chen and Zhu, Menglong and Korattikara, Anoop and Fathi, Alireza and Fischer, Ian and Wojna, Zbigniew and Song, Yang and Guadarrama, "Speed/accuracy trade-offs for modern convolutional object detectors Jonathan," *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7312–7319, 2017.

[69] A. Tonioni, "tf-objdetector." `https://github.com/AlessioTonioni/tf-objdetector`, 2018.

[70] E. Hanna and M. Cardillo, "Faster RCNN," *Biological Conservation*, vol. 158, pp. 196–204, 2013.

[71] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-Decem, pp. 770–778, 2016.

[72] J. Redmon, "Darknet: Open source neural networks in c." `http://pjreddie.com/darknet/`, 2013–2016.

[73] A. W. Services, "¿qué es docker?." Disponible en: `https://aws.amazon.com/es/docker/`, 2020.

[74] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*, pp. 740–755, Springer, 2014.

[75] AlexeyAB, "Alexeyab/darknet. git code." Disponible en: `https://github.com/AlexeyAB/darknet#how-to-improve-object-detection`.

[76] T. Rabbani, F. Van Den Heuvel, and G. Vosselmann, "Segmentation of point clouds using smoothness constraint," *International archives of photogrammetry, remote sensing and spatial information sciences*, vol. 36, no. 5, pp. 248–253, 2006.

[77] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[78] S. I. Grossman, *Álgebra lineal*. 7 ed., 2012.

[79] Intel, "Intel realsense depth camera d435i." Disponible en: `https://store.intelrealsense.com/buy-intel-realsense-depth-camera-d435i.html`, 2021.

[80] OpenCV, "Opencv ai kit: Oak-d." Disponible en: `https://store.opencv.ai/products/oak-d`, 2021.

[81] Intel, "realsense2_camera. git code." http://wiki.ros.org/realsense2_camera, 2021.

[82] Tossy, "Yolo v4 for darknet_ros. git code." https://github.com/Tossy0423/yolov4-for-darknet_ros, 2020.

[83] TNTWEN, "Openvino-yolov4. git code." Disponible en: `https://github.com/TNTWEN/OpenVINO-YOLOV4`, 2021.

[84] OpenVINO, "Openvino™ toolkit overview." Disponible en: `https://docs.openvinotoolkit.org/latest/index.html`, 2021.

[85] OpenCV, "Opencv ai kit crash course." Disponible en: `https://opencv.org/courses/`, 2021.

[86] Luxonis, "depthai-ros. git code." Disponible en: `https://github.com/luxonis/depthai-ros`, 2021.

[87] J. A. Sánchez-Rojas, J. A. Arias-Aguilar, H. Takemura, and A. E. Petrilli-Barceló, "Staircase detection, characterization and approach pipeline for search and rescue robots," *Applied Sciences*, vol. 11, no. 22, p. 10736, 2021.

[88] C. Robotics, “Simulating husky.” Disponible en: http://www.clearpathrobotics.com/assets/guides/melodic/husky/SimulatingHusky.html, 2021.