Universidad Zaragoza
1542

Trabajo Fin de Grado

Estimación de la pose de una cámara monocular con robustez frente a objetos dinámicos.

Monocular camera pose estimation robust to dynamic objects.

Autor

Víctor Sisqués Cortés

Directora

Berta Bescós Torcal

Codirector

José Neira Parra

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2020

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Víctor Sisqués Cortés,

con nº de DNI 73162425-T en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Título en español: Estimación de la pose de una cámara monocular con robustez frente a objetos dinámicos.

Título en inglés: Monocular camera pose estimation robust to dynamic objects.

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 27 de Julio de 2020

Fdo: 

AGRADECIMIENTOS

En primer lugar agradecer a mi tutora, Berta Bescós Torcal, por su ayuda y consejo además de por confiar en mí para llevar a cabo su propuesta.
Agradecer también a Alberto, Alex, Andrés, Bruno, Carlos, Nacho, Paul, Rebecca, Saúl, Sergio y Sonia.
Dar las gracias a la mayor parte de los profesores que me han dado clase hasta el día de hoy, puesto que sus enseñanzas me han ayudado a llegar hasta aquí.
Finalmente me gustaría agradecer a mis padres y a mi hermano por apoyar todas las decisiones que he tomado.

Estimación de la pose de una cámara monocular con robustez frente a objetos dinámicos.

RESUMEN

En problemas de localización visual de sistemas móviles (robots, coches autónomos, dispositivos de realidad virtual y aumentada, *etc.*) frecuentemente se hace la suposición de que la escena es completamente estática y que el efecto de los objetos dinámicos es despreciable. Sin embargo, esto limita el uso de estos sistemas en escenas pobladas del mundo real como carreteras con alto tráfico en el caso de coches autónomos, u hogares en el caso de robots de servicio. La presencia de objetos dinámicos degrada la precisión de la estimación de los seis grados de libertad de la pose de la cámara. El objetivo de este trabajo de fin de grado consiste en detectar regiones dinámicas en las imágenes de una escena en la que haya un contenido dinámico significativo para así estimar con precisión y robustez la pose relativa entre dos imágenes consecutivas de una secuencia.

Para abordar este problema se van a utilizar tanto técnicas clásicas de visión por computador como técnicas de aprendizaje profundo. Más concretamente se ha hecho un estudio de la distribución de *inliers* y *outliers* del modelo clásico de estimación de la pose de una cámara en escenas altamente dinámicas, utilizando detección y emparejamiento de puntos de interés, cálculo de la matriz fundamental mediante RANSAC, *etc.* En cuanto a técnicas de aprendizaje profundo, se ha aprovechado el uso de la segmentación semántica para poder tener una comprensión de la escena observada a alto nivel y poder razonar a nivel de objetos en vez de a nivel de puntos del mapa. La combinación de ambas técnicas nos permite obtener una mejor distribución de *inliers* y *outliers* a un modelo de escena estático, así como poder descubrir objetos dinámicos y estáticos en la escena observada.

Monocular camera pose estimation robust to dynamic objects.

ABSTRACT

In visual localization systems of moving elements (autonomous robots and vehicles, agumented and virtual reality devices, *etc.*) The assumption is often made that the scene is completely static, or that the effect of dynamic objects is negligible. However, this limits the use of these systems in populated real-world scenes, such as high-traffic roads in case of autonomous cars or homes in case of service robots. The presence of dynamic objects degrades the accuracy of the estimation of the six degrees of freedom of the camera's pose. The objective of this bachelor thesis is to detect dynamic regions in the images of a scene in which there is dynamic content so that one can estimate with high precision and robustness the relative pose between two consecutive images in a sequence.

To address this problem, both classical computer vision techniques and deep learning techniques will be used. More specifically, a study of the distribution of *inliers* and *outliers* of the classical model for camera pose estimation in highly dynamic scenes, using feature detection and keypoint matching, calculation of the fundamental matrix using RANSAC, *etc.* Regarding deep learning techniques, the use of semantic segmentation has been used to be able to have an understanding of the observed scene and to be able to reason at the level of objects instead of at the level of map points. The combination of both techniques allows us to obtain a better distribution of *inliers* and *outliers* to a static scene model, as well as being able to discover dynamic objects in the observed scene.

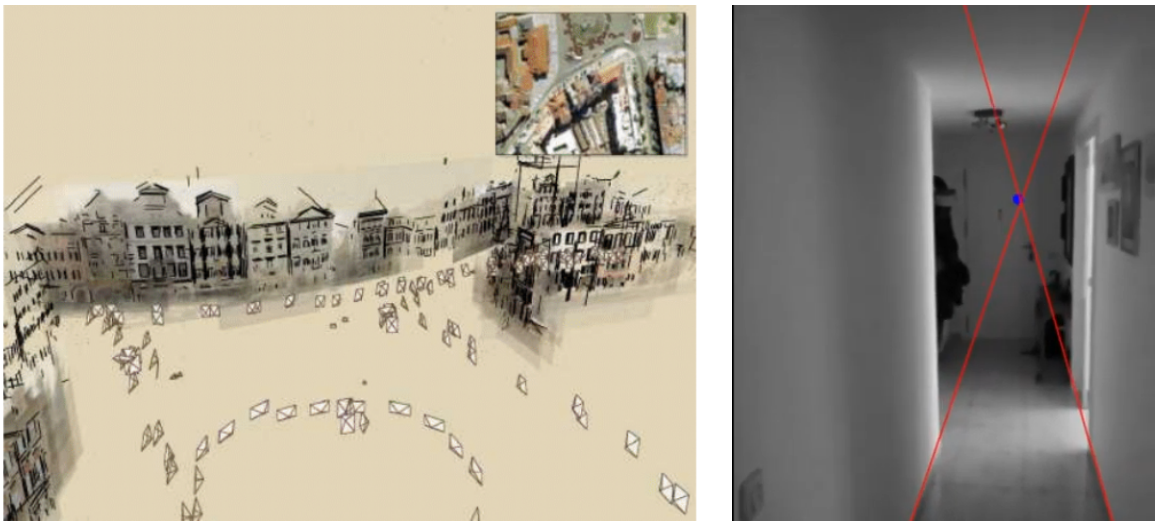
Índice

Capítulo 1

Introducción

1.1. Visión por Computador

La visión por computador o visión artificial es una disciplina científica que incluye métodos para adquirir, procesar, analizar y comprender las imágenes del mundo real con el fin de producir información que pueda ser tratada por un ordenador. La visión por computador trata de conseguir que los ordenadores puedan percibir y comprender una imagen o secuencia de imágenes y actuar según convenga en una determinada situación. Esta comprensión se consigue gracias a distintos campos como la **geometría**, la **estadística**, la **física** y otras disciplinas [1].



(a) Reconstrucción 3D llevada a cabo por el sistema propuesto por Snavely *et al.* [2]

(b) Estimación del punto de fuga de una escena.

Figura 1.1: Ejemplos de sistemas de visión por computador.

Hay muchas tecnologías que utilizan la visión por computador (Figura 1.1), como por ejemplo sistemas para estimar el punto de fuga de una escena (Figura 1.1b) o sistemas de reconstrucción 3D (Figura 1.1a), esta última se engloba dentro de la visión 3D, que es la disciplina que nos incumbe en este proyecto. La visión 3D se

encarga de proporcionar a los ordenadores la capacidad de emular la visión humana, con esta capacidad el ordenador podrá generar un modelo tridimensional de un objeto o escena, generalmente a partir de una secuencia de imágenes bidimensionales tomadas con un sistema monocular. También existen técnicas y configuraciones de sensores que permiten conocer la geometría 3D de una escena con una sola muestra: sistemas estereoscópicos, sistemas RGB-D, *etc.*, pero estos no van a ser estudiados en este trabajo de fin de grado.

1.2. Geometría 3D

La reconstrucción en 3D a partir de múltiples imágenes busca averiguar la geometría de una escena capturada por una colección de imágenes (Figura 1.1a). Por lo general, los parámetros internos y la posición de la cámara son conocidos o se pueden estimar a partir del conjunto de imágenes. A la hora de estimar la posición de la cámara, es necesario que se cumplan una serie de hipótesis acerca de la escena, como pueden ser la consistencia de la iluminación, la nitidez de las imágenes, la estaticidad de la escena, *etc.* En concreto, este trabajo se va a centrar en estudiar la influencia que tienen los elementos dinámicos a la hora de estimar la posición de la cámara, es decir, de aquellos elementos que se mueven en una dirección o velocidad distinta a la que se mueve la escena estática. Puede verse en la Figura 1.2 que el camión blanco es un objeto dinámico. Los algoritmos tradicionales de visión por computador de estimación de pose podrían verse influenciados por la presencia de este objeto, ya que ante la suposición de una escena estática el movimiento de este objeto podría calcularse incorrectamente como el propio movimiento de la cámara [3].

Figura 1.2: Ejemplo de un objeto dinámico que entorpece el proceso de estimación de la posición de la cámara puesto que se mueve en dirección opuesta al resto de elementos. Estas imágenes provienen del *dataset* de conducción autónoma KITTI [4].

El cálculo de la posición de la cámara en entornos desconocidos es la base de sistemas de localización que son utilizados en aplicaciones de realidad aumentada (RA), de conducción autónoma, y de realidad virtual (RV) entre otros. Si este cálculo no se realiza correctamente, surgen problemas que pueden llegar a provocar el malestar de los usuarios, como mareos en el caso de la RA y la RV, e incluso amenazas para la seguridad de las personas como colisiones en sistemas de conducción autónoma. El principal objetivo de este proyecto es estudiar la repercusión de la presencia de objetos dinámicos a la hora de estimar la posición de una cámara en los algoritmos tradicionales, y la proposición de una red neuronal capaz de diferenciar entre objetos dinámicos y estáticos para así reducir la influencia de los primeros en algoritmos de localización.

Este proyecto ha sido propuesto por mi directora Berta Bescós en un contexto de investigación para sistemas de localización y mapeo como puede ser el SLAM. Asumir que la escena observada es estática es común en los algoritmos de SLAM visual. Esta suposición es válida para algunas aplicaciones pero limita su utilidad en escenas concurridas del mundo real para la conducción autónoma, los robots de servicio o realidad aumentada y virtual entre otros. La detección y el estudio de objetos dinámicos es un requisito para estimar con precisión la posición del sensor y construir mapas estables, útiles para aplicaciones robóticas que operan a largo plazo.

1.3. Organización del Documento

Las tecnologías utilizadas se explican detalladamente en el Capítulo 2, para así tener una base teórica sobre el tema que permita comprender este proyecto en su totalidad.

Para estudiar la repercusión de la presencia de objetos dinámicos en una escena se van a utilizar diversas técnicas de visión por computador tradicional, como pueden ser detectores de puntos de interés y estimadores de pose. Este estudio se realizará comparando los valores de posición proporcionados por los *datasets* con los valores de posición calculados por el algoritmo desarrollado. Además se va a comprobar la precisión con la que un sistema tradicional es capaz de detectar objetos dinámicos. El funcionamiento y los resultados de este estudio se encuentran en el Capítulo 3.

La red neuronal que se ha desarrollado para este TFG tiene como objetivo determinar los objetos dinámicos en una escena dadas 2 imágenes RGB de esta misma escena. Esta red busca detectar los objetos dinámicos con mayor precisión que la conseguida en un sistema tradicional. El proceso de entrenamiento, la estructura de la red, el *dataset* utilizado para entrenar y los resultados se explican en el Capítulo 4.

Capítulo 2

Fundamentos

En este proyecto se han utilizado diferentes tecnologías dentro del campo de la visión por computador, a continuación se van a explicar estas tecnologías y conceptos más detalladamente para así asentar una base de conocimiento de visión por computador.

Las tecnologías descritas se dividen en técnicas tradicionales de visión por computador y redes neuronales aplicadas a visión, esta división se ha hecho así puesto que los contenidos mostrados en la primera mitad tienen una base teórica más sencilla de comprender y estos conocimientos se ven recogidos en varios libros de visión por computador clásica, mientras que las redes neuronales son mucho más recientes que las técnicas recogidas en estos libros [5].

2.1. Técnicas tradicionales

2.1.1. Puntos de interés

Los puntos de interés, características, *keypoints* o *features* son patrones locales de una imagen que difieren de sus vecinos. Hay varios tipos de puntos de interés, siendo los más importantes los puntos de interés de tipo esquina y los puntos de interés de tipo *blob*. Estos últimos tienen como característica que son invariantes a escala, por lo que son muy utilizados para el reconocimiento de objetos. Los puntos de interés son usados principalmente para realizar emparejamiento de imágenes, esto nos permite conseguir estabilizar vídeos, hacer *tracking* de objetos, generar panoramas dado un conjunto de imágenes de la misma escena. También pueden utilizarse para realizar reconocimiento de objetos y lugares [6]. Los aspectos más importantes de los puntos de interés son su repetibilidad y su robustez, es decir, lo óptimo es que una característica pueda ser encontrada en varias imágenes de una escena y no se confunda con otra [5].

Entre los detectores de puntos de interés destacan los siguientes tipos.

– Detectores de esquinas. Destacan Harris, Shi-Tomasi y FAST [7, 8, 9].

– Detectores invariantes a escala. Destacan SIFT y SURF [6, 10].

2.1.2. Emparejamiento de puntos de interés

Una vez obtenidos los puntos de interés de una imagen, se pasa a calcular los descriptores de estos. Los descriptores son vectores que codifican información sobre los puntos de interés y nos permiten diferenciarlos entre sí, destacan descriptores binarios como BRIEF y ORB [11, 12]. Idealmente la información de los descriptores es invariante a transformaciones de la imagen.

El proceso de emparejamiento de puntos de interés se basa en, dados los descriptores de dos imágenes, compararlos mediante fuerza bruta para encontrar las similitudes entre estos. En la Figura 2.1 se muestra un ejemplo de un sistema de emparejamiento de puntos de interés [5]. Se puede observar que a pesar de que los puntos que se emparejan han sufrido una transformación de escala y de rotación, los descriptores utilizados (ORB [12]) abstraen la información de la rotación y de la escala hasta un cierto punto y permiten un emparejamiento correcto.

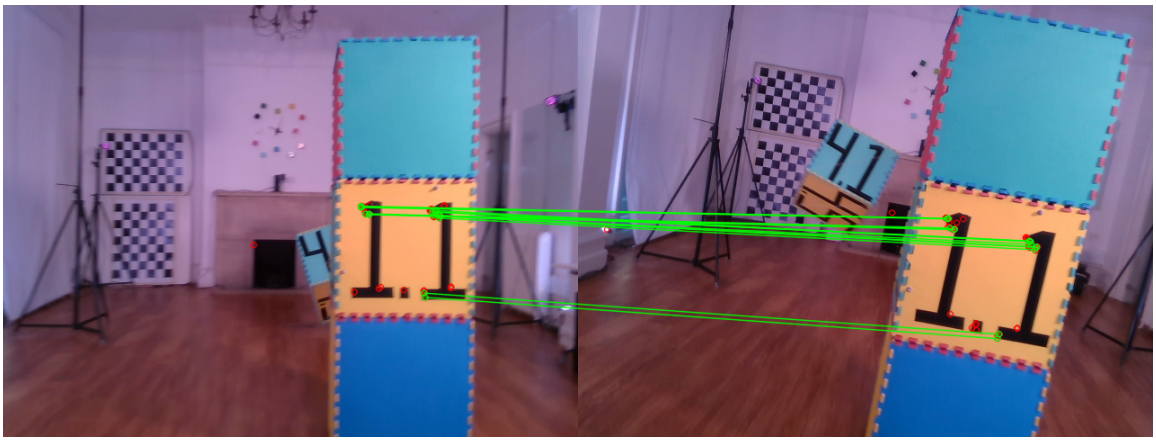


Figura 2.1: Ejemplo de de emparejamiento de puntos de interés.

2.1.3. Flujo óptico

El flujo óptico es el patrón del movimiento aparente de los objetos de una escena entre dos *frames*, causado por el movimiento relativo entre un observador (un ojo o una cámara) y la escena. Es un campo de vectores bidimensionales donde cada vector codifica un desplazamiento mostrando el movimiento de puntos desde el primer *frame* al segundo (Figura 2.2).

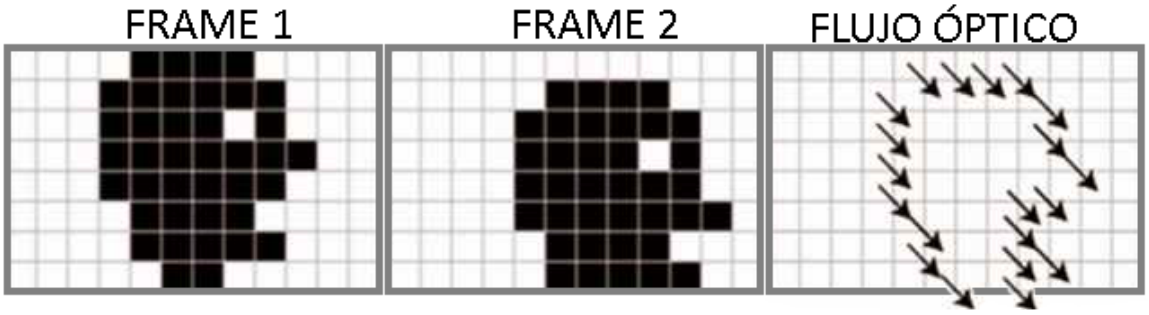


Figura 2.2: Ejemplo de flujo óptico.

Para la estimación del flujo óptico es común hacer las siguientes asunciones:

- La intensidad de los píxeles de un objeto no cambian entre *frames* consecutivos.
- Todos los píxeles vecinos a un píxel dado, en un mismo objeto, tienen un movimiento similar

Dado un píxel con intensidad $I(x, y, t)$ en el primer *frame* (la tercera dimensión representa el tiempo), que se mueve una distancia (dx, dy) en el próximo *frame* tras haber pasado dt tiempo, se puede decir que,

$$I(x, y, t) = I(x + dx, y + dy, t + dt) \tag{2.1}$$

Luego, tras aproximar el lado derecho mediante la serie de Taylor[1], eliminar términos comunes y dividir entre dt se obtiene la siguiente ecuación:

$$f_z u + f_y v + f_t = 0, \tag{2.2}$$

donde:

$$f_x = \frac{\partial f}{\partial x}; \quad f_y = \frac{\partial f}{\partial y} \tag{2.3}$$

$$u = \frac{dx}{dt}; \quad v = \frac{dy}{dt} \tag{2.4}$$

La Ecuación 2.2 es la llamada ecuación del Flujo Óptico. De esta ecuación se conocen f_x y f_y que son gradientes de la imagen, y f_t que es el gradiente temporal, y son desconocidos (u, v), por lo que no se puede resolver esta ecuación. Debido a esto han aparecido varios métodos para obtener el resultado, entre los que destacan el método de Lucas-Kanade [14, 15].

[1]Serie de taylor — Wikipedia, The Free Encyclopedia [13]

2.1.4. Matriz fundamental y matriz esencial

La matriz fundamental es una relación entre dos imágenes de la misma escena (imágenes estéreo) que restringe dónde puede ocurrir la proyección de puntos de la escena en ambas imágenes. Dada la proyección de un punto en una de las imágenes, el punto correspondiente en la otra imagen se limita a una línea. Supongamos los puntos $\mathbf{x} = [u, v, 1]^T, \mathbf{x}' = [u', v', 1]^T \in \mathbb{R}^3$ pertenecientes a dos imágenes distintas de la misma escena, y la matriz $\mathbf{F} \in \mathbb{R}^{3\times3}$. $\mathbf{F}$ será la matriz fundamental de la escena si para todos los puntos de esta se cumple

$$(\mathbf{x}')^T\mathbf{F}\mathbf{x} = 0. \tag{2.5}$$

La matriz esencial $\mathbf{E} \in \mathbb{R}^{3\times3}$ se basa en el mismo principio que la matriz fundamental pero solo se cumple cuando las cámaras utilizadas satisfacen el modelo de cámara estenopeica (*pinhole camera model*).

El modelo de cámara estenopeica describe las relaciones matemáticas entre las coordenadas de un punto en un espacio tridimensional y su proyección en el plano imagen de la cámara. La apertura de este modelo de cámaras se describe como un punto y no utiliza ningún tipo de lentes para enfocar la luz. Este modelo solo puede ser utilizado como aproximaciones del mapeo entre una escena tridimensional y una bidimensional [16].

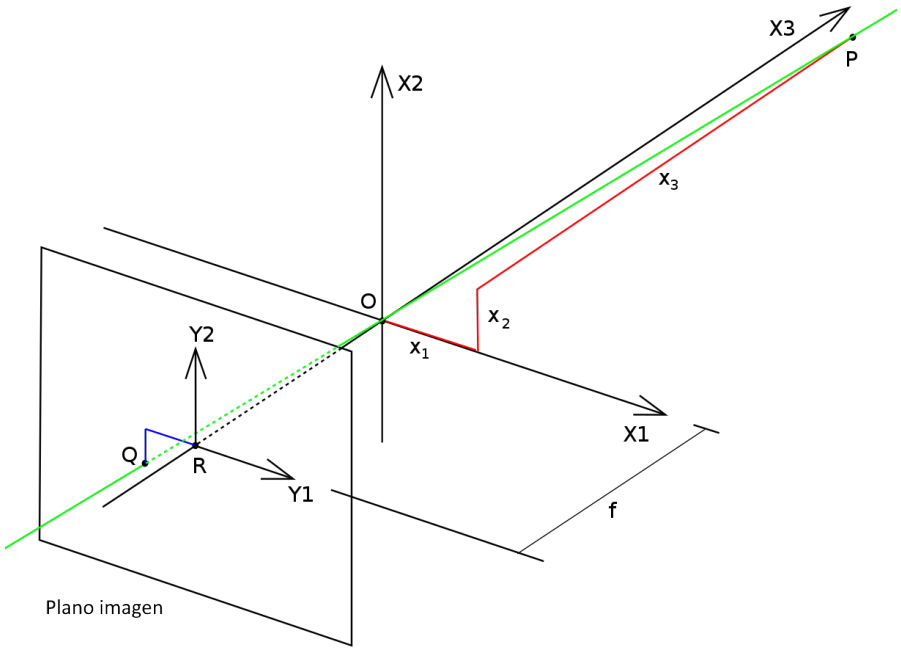


Figura 2.3: Representación de la geometría de una cámara *pinhole* [16].

La geometría relacionada con el mapeo de una cámara *pinhole* se ilustra en la Figura 2.3. Esta figura contiene los siguientes objetos:

– Un sistema ortogonal de coordenadas 3D con su origen en **O**. **O** también representa el punto de apertura de la cámara o centro óptico. Los tres ejes de coordenadas son $\mathbf{X_1}$, $\mathbf{X_2}$ y $\mathbf{X_3}$. Este último apunta en la dirección de visión de la cámara, también referido como el eje óptico. El plano formado por $\mathbf{X_1}$ y $\mathbf{X_2}$ es la parte frontal de la cámara.

– Un plano imagen, donde se proyecta el mundo 3D a través de la apertura de la cámara. El plano imagen es paralelo a los ejes $\mathbf{X_1}$ y $\mathbf{X_2}$ y se encuentra a una distancia f del origen **O** en dirección negativa del eje $\mathbf{X_3}$, donde f es la distancia focal de la cámara *pinhole*.

– Un punto **R** en la intersección del eje óptico y el plano imagen. Este punto se conoce como punto principal.

– Un punto **P** en el mundo con coordenadas $(\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3})$ relativas a los ejes $\mathbf{X_1}$, $\mathbf{X_2}$ y $\mathbf{X_3}$.

– La linea que proyecta el punto **P** a la cámara, linea de proyección (linea verde).

– La proyección del punto **P** en el plano imagen, **Q**. Este punto esta definido por la intersección de la linea de proyección (linea verde) y el plano imagen.

– El plano imagen tiene también un sistema de coordenadas bidimensional propio, con origen en **R** y ejes $\mathbf{Y_1}$ y $\mathbf{Y_2}$ paralelos a $\mathbf{X_1}$ y $\mathbf{X_2}$, respectivamente. Las coordenadas de **Q** en este sistema de coordenadas son $(\mathbf{y_1}, \mathbf{y_2})$

Los parámetros de la cámara *pinhole* se ven representados en la matriz de calibración intrínseca **K** (Ecuación 2.6), siendo estos parámetros la distancia focal (f_x, f_y) y el centro óptico (c_x, c_y) medidos en los ejes x e y respectivamente.

$$\mathbf{K} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{2.6}$$

La relación entre la matriz fundamental y la esencial establece que

$$\mathbf{E} = (\mathbf{K}')^T \mathbf{F} \mathbf{K} \tag{2.7}$$

donde $\mathbf{K}, \mathbf{K}' \in \mathbb{R}^{3\times3}$ son las matrices de calibración intrínseca de las dos imágenes involucradas. Una vez obtenida la matriz esencial, se pueden aplicar diversas operaciones a esta para poder obtener la posición relativa de la cámara entre las dos imágenes de la misma escena. El cálculo de las matrices comentadas en este apartado suelen realizarse mediante el método de RANSAC (Subsección 2.1.5) [17, 18].

2.1.5. RANSAC

Random sample consensus o RANSAC es un método iterativo para estimar parámetros de un modelo matemático dado un conjunto de datos que contiene valores atípicos (a partir de ahora referidos como *outliers*). Es un algoritmo no determinista, ya que produce un resultado razonable sólo con cierta probabilidad, aumentando esta probabilidad conforme aumenta el número de iteraciones que ejecuta este algoritmo.

RANSAC asume que los datos consisten de valores cuya distribución puede ser explicada por un conjunto de parámetros del modelo (a partir de ahora referidos como *inliers*) o *outliers*. Los *outliers* pueden provenir de errores de medición o ruido, por ejemplo. RANSAC también asume que, dado un pequeño conjunto de *inliers*, existe un procedimiento que puede estimar los parámetros de un modelo que explica de manera óptima estos datos.

En este estudio se consideran *outliers* todos aquellos elementos dinámicos, puesto que para un cálculo correcto de la matriz fundamental todos los *inliers* deberían ser elementos estáticos [19].

2.1.6. Simultaneous Localization And Mapping (SLAM)

El problema de localización y construcción visual simultánea de mapas (visual SLAM por sus siglas en inglés *Simultaneous Localization and Mapping*) consiste en localizar una cámara en un mapa que se construye en tiempo real. Esta tecnología permite la localización de robots en entornos desconocidos y la creación de un mapa de la zona con los sensores que lleva incorporados, es decir, sin contar con ninguna infraestructura externa [20]. A diferencia de los enfoques de odometría en los cuales el movimiento incremental es integrado en el tiempo, un mapa permite que el sensor se localice continuamente en el mismo entorno sin acumular deriva.

Un sistema de SLAM a destacar es ORB-SLAM [21], desarrollado por la Universidad de Zaragoza, que implementa un extractor de puntos de interés de tipo ORB que será usado en este trabajo.

2.2. Redes neuronales aplicadas a visión

2.2.1. Redes neuronales

Una red neuronal es una representación de datos en capas (Figura 2.4) capaz de aprender de los datos que se le proporcionan para generar salidas con sentido. Las redes neuronales a la hora de procesar los datos, los representa de diferentes maneras y dimensiones mediante la aplicación de operaciones específicas para transformar nuestros

datos en cada capa. Al realizar estas transformaciones, la red puede entender mejor nuestros datos y, por lo tanto, proporcionar mejores predicciones.

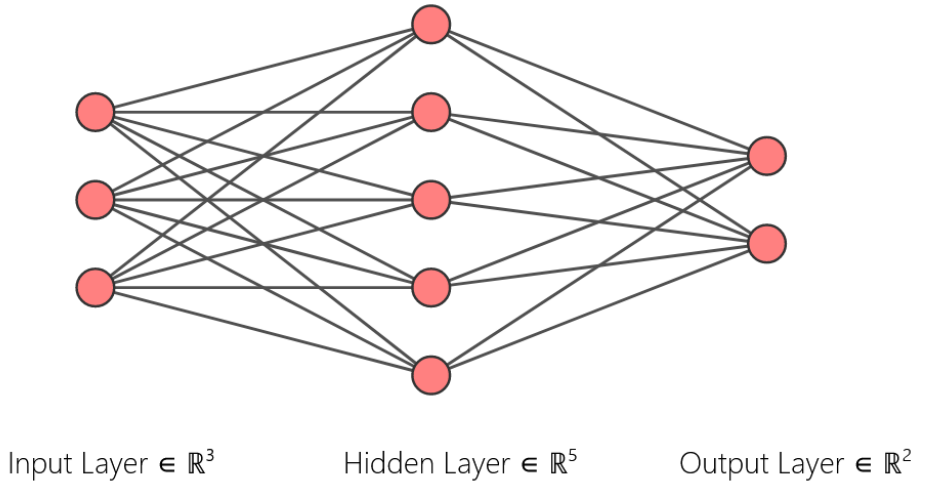


Figura 2.4: Ejemplo de red neuronal.

A más bajo nivel, las redes neuronales son simplemente una combinación de operaciones matemáticas y álgebra lineal. Cada red está formada por una secuencia de capas a través de las cuales pasan los datos. Estas capas están formadas por neuronas y las neuronas de una capa se conectan con la siguiente (capas densas). Estas conexiones se definen por los pesos. Además de los pesos individuales, cada capa tiene un sesgo (*bias* en inglés) que es un valor numérico constante. Los datos entran por la capa de entrada, se transforman conforme pasan por las siguientes capas y se devuelve una predicción en la capa de salida.

La salida de una neurona es el sumatorio de los datos de salida de todas las neuronas anteriores:

$$\mathbf{Y} = F((\sum_{i=0}^{n} w_i x_i) + b), \tag{2.8}$$

donde:

- Y es la salida de la neurona.
- F es la función de activación de la capa.
- w es el peso de cada conexión a esta neurona.
- x es el valor que proviene de la capa anterior.
- b es el *bias* de cada capa.
- n es el número de conexiones.

Al principio del entrenamiento, y tras especificar las funciones de activación de cada capa, los pesos y *bias* de cada capa se inicializan de manera aleatoria o siguiendo algún tipo de función de distribución predeterminada. Conforme se van pasando datos a la red, esta va aprendiendo los pesos y *bias* correctos y ajusta la red en consecuencia usando retro-propagación. La retro-propagación o *backpropagation* es el algoritmo fundamental detrás del proceso de entrenamiento, y es el encargado de determinar las combinaciones de pesos que mejor resultado nos dan. Para comprender este proceso es necesario conocer las funciones de coste y los optimizadores.

- Las funciones de coste son funciones responsables de determinar cuan bien ha predicho los resultados la red neuronal, comparando el resultado obtenido con el resultado esperado y nos devuelve un valor representando el coste de la red.

- Los optimizadores son las diferentes funciones que pueden implementar el algoritmo de retro-propagación. Entre estas funciones se pueden encontrar el descenso de gradiente, el descenso de gradiente estocástico y *ADAM* (*Adaptative Movement Estimator*). Este último es el más utilizado, puesto que converge rápidamente a costa de un mayor coste computacional. La calidad del optimizador suele depender del ratio de aprendizaje, que es un parámetro que determina la velocidad con la que el algoritmo convergerá; un ratio de aprendizaje muy elevado puede implicar que el algoritmo no converja mientras que un ratio de aprendizaje muy pequeño implica mayor tiempo de ejecución necesario para alcanzar la convergencia (Figura 2.5).

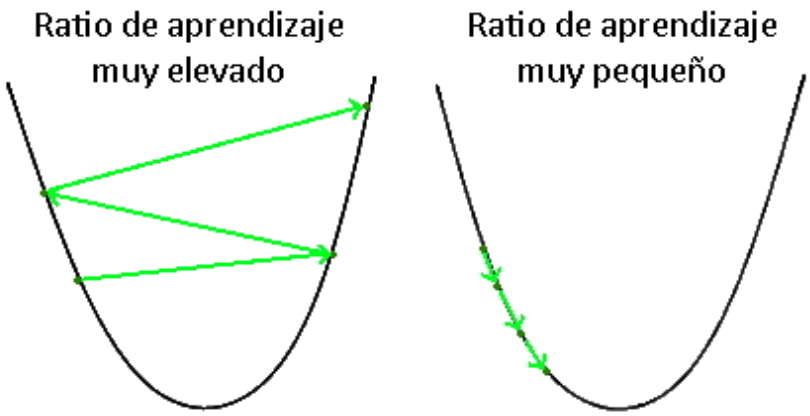


Figura 2.5: Implicaciones de un ratio de aprendizaje muy grande y muy pequeño.

El objetivo del algoritmo de retro-propagación es minimizar el coste que devuelve la función de coste utilizando un optimizador específico [22].

2.2.2. Sobreajuste y subajuste

A la hora de entrenar una red neuronal, se busca que esta aprenda de un conjunto de datos determinado con el objetivo de que lo aprendido sirva para cualquier dato de entrada.

En ocasiones, la red neuronal se adapta demasiado a los datos con los que entrena empeorándose así los resultados obtenidos con los datos que no aparecen en este conjunto, este fenómeno se denomina sobreajuste. El subajuste se da en casos en los que el entrenamiento no es lo suficientemente bueno y la red no aprende, provocando esto malos resultados para cualquier dato de entrada.

En la Figura 2.6 se ilustran casos de sub y sobreajuste.

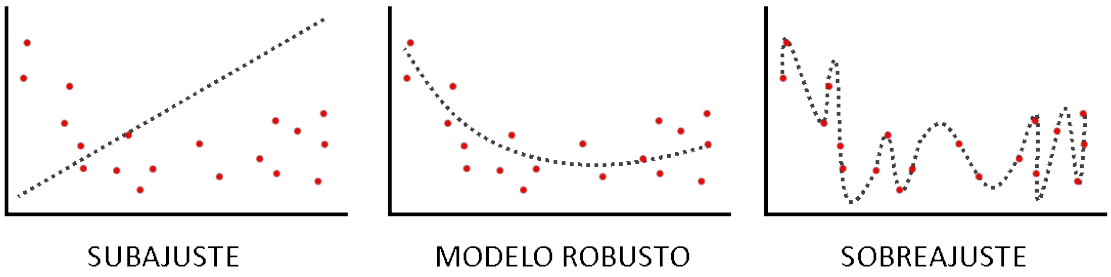


Figura 2.6: Ejemplo de un modelo entrenado con subajuste, uno entrenado correctamente y otro con sobreajuste.

Para evitar el subajuste es necesario proporcionar más datos de entrenamiento; en cambio, para evitar el subajuste se pueden usar técnicas como la regularización o añadir una capa de *dropout*, que elimina conexiones entre neuronas con una cierta probabilidad dada.

2.2.3. Redes neuronales convolucionales

Las redes neuronales convolucionales (RNC a partir de ahora) están compuestas por una o más capas convolucionales. Estas capas son distintas a las convencionales, puesto que su objetivo es encontrar patrones en imágenes de manera local.

Las capas densas, a la hora de trabajar con imágenes son capaces de aprender patrones que aparecen en una zona en especifico de la imagen. Esto implica que si un patrón que la red conoce aparece en una zona distinta a donde lo ha aprendido, tendrá que reaprender este patrón para poder detectarlo. En cambio, las capas convolucionales aprenden los patrones sin tener en cuenta dónde aparecen, por lo que si se aprende un patrón en concreto este será identificado correctamente independientemente de dónde aparezca.

Las capas convolucionales aplican filtros a los valores de entrada. Un filtro es un patrón de m píxeles que se busca en los valores de entrada. El número de filtros

representa el número de patrones que se buscan en cada capa. En la Figura 2.7 se muestra un ejemplo de como funcionan los filtros [22].

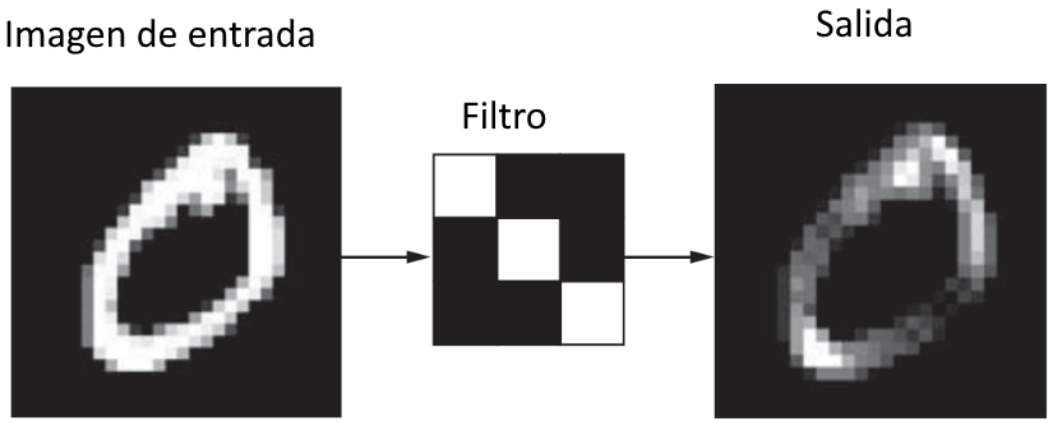


Figura 2.7: Aplicación de un filtro a una imagen. Adaptación de una imagen proveniente del libro *Deep Learning with Python*[22].

2.2.4. Segmentación semántica

La segmentación semántica hace referencia al proceso de asignar a cada píxel de una imagen una etiqueta de clase (Figura 2.8). Se puede pensar este proceso como clasificación de imágenes a nivel de píxel. Por ejemplo, en una imagen en la que aparece un número elevado de coches, la segmentación etiquetará estos objetos como coches; si se busca determinar diferentes instancias de una misma clase se hace referencia a segmentación de instancias.

La segmentación semántica es muy importante en robots y vehículos autónomos, ya que estos necesitan comprender el contexto del entorno en el que se encuentran.

En este proyecto se explora el uso de Mask-RCNN [23] y YOLACT [24], que son el actual estado del arte en cuanto a redes de segmentación semántica respecta.

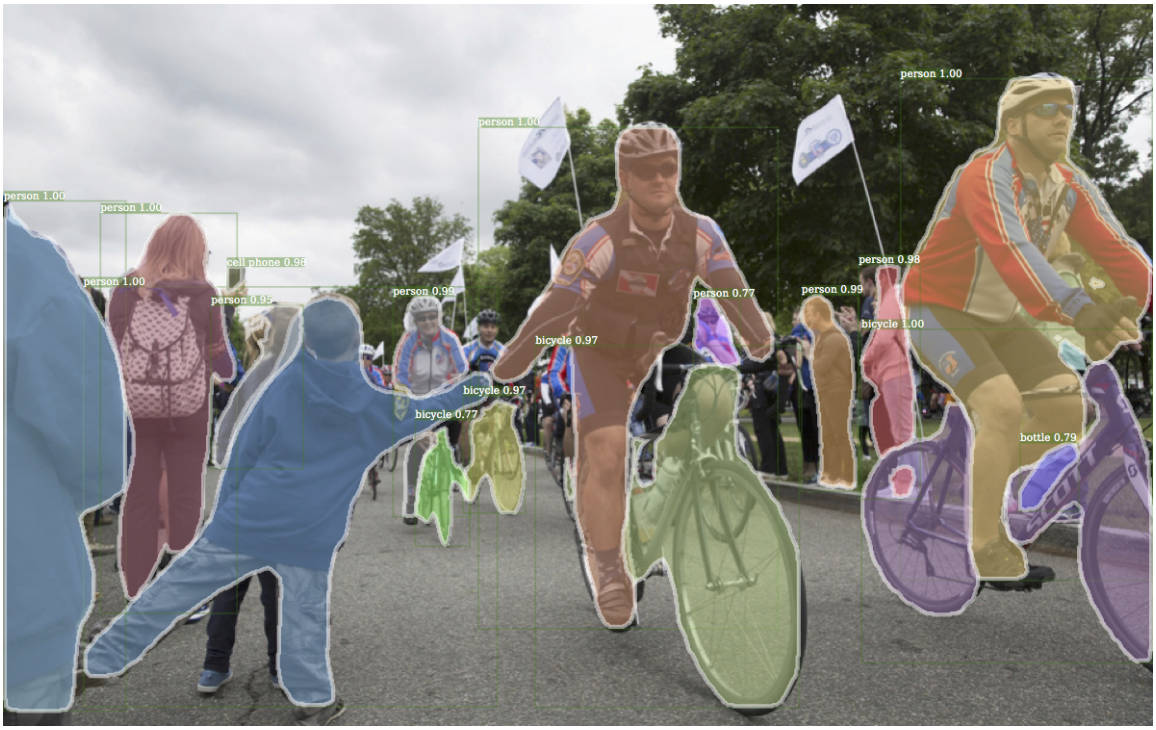


Figura 2.8: Resultado de aplicar segmentación semántica a una imagen [23]

2.2.5. TensorFlow

TensorFlow es una biblioteca de código abierto para aprendizaje automático desarrollado por Google, programado en Python y C++ proporciona una API para un gran número de lenguajes de programación. Es una biblioteca centrada en el *dataflow*[2] y *differentiable programming*[3], utilizada principalmente en aplicaciones de aprendizaje automático.

El objeto principal utilizado por TensorFlow son los tensores, una generalización de los vectores y matrices a dimensiones superiores, TensorFlow los representa internamente como vectores n-dimensionales. Los tensores son transmitidos y manipulados por el programa; cada tensor representa una computación definida parcialmente que eventualmente producirá un valor. Los programas de TensorFlow funcionan creando un grafo de tensores que detalla las relaciones entre estos, ejecutar diferentes partes del grafo nos proporciona los resultados.

[2]Dataflow programming — Wikipedia, The Free Encyclopedia [25]
[3]Differenntiable programming — Wikipedia, The Free Encyclopedia [26]

Capítulo 3

Inliers y Outliers al Modelo de Escena Estática

Para realizar el estudio sobre la repercusión de objetos dinámicos en una escena, se ha utilizado el *dataset* de objetos dinámicos de Oxford Multimotion Dataset [27]. Más concretamente se han usado las secuencias *occlusion 2 unconstrained* y *swinging 4 unconstrained*. Estos *datasets* han sido grabados en habitaciones cerradas con una cámara estéreo, una cámara de profundidad y una cámara RGB. En estas habitaciones hay un conjunto de objetos rígidos móviles de los cuales se conoce su posición en todo momento. La posición y orientación de las cámaras también es conocida a lo largo de las secuencias, siendo esta monitorizada mediante un sistema de captura de movimiento.

El objetivo de utilizar este dataset es comprobar si al calcular la matriz fundamental del movimiento de la cámara (Subsección 2.1.4) mediante RANSAC (Subsección 2.1.5) para dos imágenes de una misma secuencia, los *outliers* resultantes tienden a aparecer en los objetos dinámicos, ya que si la matriz fundamental está bien calculada sólo debería usar emparejamientos pertenecientes a objetos estáticos como *inliers*. Para calcular la precisión y exhaustividad de este método de detección es necesario disponer de las máscaras binarias de los objetos dinámicos para así poder filtrar correctamente aquellos puntos de interés que son realmente dinámicos.

Teniendo esto en cuenta, la teoría dice que los *outliers*, a parte de en zonas con patrones repetitivos o con ruido, deberían aparecer de forma mayoritaria en los objetos dinámicos, por lo que en este capítulo se va a comprobar la efectividad de este algoritmo tradicional de visión por computador para detectar elementos dinámicos. Al llevar estas suposiciones a la práctica se ve que no es completamente así debido a la existencia de ruido a la hora de estimar la posición mediante RANSAC y a que la aparición de los objetos dinámicos es en muchos casos de las secuencias a estudiar muy significativa.

3.1. Detección de puntos de interés

Para detectar puntos de interés en las imágenes a tratar se han utilizado diferentes métodos, pero finalmente se ha decidido por el uso del extractor ORB implementado en ORB-SLAM [21], debido a su rapidez, poco uso de memoria y robustez frente a diferentes orientaciones y escalas (Figura 3.1).

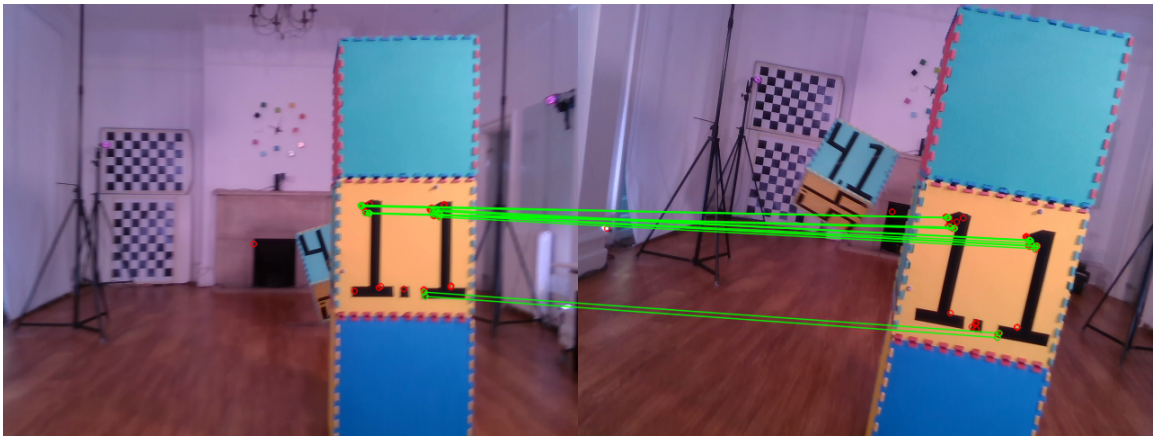


Figura 3.1: Resultado de emparejar los puntos de interés calculados para 2 *frames*, se observa que a pesar de haber sido rotados, los emparejamientos son correctos.

Al principio del proyecto se utilizó el extractor de puntos de interés ORB implementado en *OpenCV*, pero se desechó ya que este extractor tiende a calcular un gran número de puntos de interés en zonas con gran número de patrones similares (como el tablero de ajedrez en la Figura 3.2), cosa que implica que muchos emparejamientos sean erróneos ya que en esas zonas los descriptores son muy similares y, como consecuencia, el cálculo de la matriz fundamental tendrá un mayor error. Este fue el principal motivo para utilizar el extractor de puntos de interés ORB implementado en ORB-SLAM [21] ya que este extractor calcula puntos de interés más dispersos en la imagen. Este extractor divide la imagen de entrada en cuadrantes y para cada cuadrante fuerza la detección de un número mínimo de puntos de interés resultando en una distribución de *features* más homogénea en las imágenes.

En la Figura 3.2 se muestran los puntos de interés extraídos para el mismo *frame* usando los dos extractores.

(a) Extractor ORB de OpenCV. (b) Extractor ORB de ORB-SLAM.

Figura 3.2: En la imagen (a) se observa que la mayor parte de los puntos de interés caen en el tablero de ajedrez del fondo, mientras que en la imagen (b) los puntos de interés, a parte de aparecer en el tablero, también están más dispersos en la imagen, por ejemplo en los trípodes de los laterales de la imagen y por el contorno del objeto dinámico.

3.2. Emparejamiento de puntos de interés y cálculo de la matriz fundamental

Una vez extraídos los puntos de interés de 2 imágenes, se debe realizar el proceso de emparejamiento; cuanto mejor sea este proceso, mejor se calculará la matriz fundamental de la cámara y el error al recuperar la pose de la cámara disminuirá. Para obtener el mejor resultado posible se ha implementado un sistema de emparejamiento robusto [28] que sigue los siguientes pasos:

1. Emparejar los puntos de interés de ambas imágenes.
2. Con el resultado, aplicar RANSAC para obtener la matriz fundamental.
3. Una vez obtenida, es utilizada para considerar los *inliers* como emparejamientos correctos.
4. Estos emparejamientos correctos serán utilizados para recalcular la matriz fundamental, esta vez usando el algoritmo determinista de los 8 puntos [29].
5. Finalmente, con esta nueva matriz fundamental, se utiliza la función "correctMatches" de *OpenCV* para refinar los emparejamientos en función de la matriz mejorada.

El código utilizado para la implementación se encuentra en la Subsección A.0.3 del anexo.

3.3. Obtención de máscaras

Una vez calculada la matriz fundamental, es necesario aplicar una máscara a *outliers* e *inliers* para así determinar la precisión y exhaustividad del sistema propuesto. A continuación se expone el método utilizado para extraer las máscaras en el *dataset* de Oxford [27].

El *dataset* de Oxford [27] está compuesto por secuencias de imágenes de cajas móviles y un *groundtruth* muy detallado sobre la posición de estas y de la cámara en todo momento. El *dataset* está capturado con una cámara estéreo y una cámara RGB-D para obtener información de profundidad además de una cámara RGB para obtener información de color. Los valores recogidos en el *groundtruth* provienen de un sistema de captura de movimiento que monitoriza en todo momento la posición de la cámara y de los objetos. Las cajas son desplazadas mediante un sistema de poleas motorizado (cajas colgantes) o mediante unas ruedas a control remoto (caja alargada).

Las cajas presentes en las secuencias de imágenes no son clasificadas correctamente por Mask-RCNN [23]. Como se observa en la Figura 3.3, Mask R-CNN clasifica el objeto dinámico como un libro con una máscara imprecisa, mientras que el de atrás ni siquiera lo clasifica. Esto se debe a que esta red neuronal no está entrenada para este tipo de objetos, por lo que, para generar las máscaras de los objetos dinámicos se va a hacer uso del *groundtruth* de las posiciones de las cámaras y de los objetos.

Figura 3.3: Resultado de aplicar Mask-RCNN al *dataset* de Oxford [27].

Otra cosa a tener en cuenta es que este *dataset* ha sido grabado con una cámara estéreo además de la cámara RGB-D, y que todos los datos almacenados en el *groundtruth* son respecto de la cámara izquierda del sistema estéreo. A continuación se muestran los pasos realizados para obtener las máscaras. Cabe destacar que se trabaja con coordenadas homogéneas.

1. Obtener la matriz de calibración de la cámara RGB, $\mathbf{K} \in \mathbb{R}^{3\times3}$ y la matriz de transformación de las cámaras estéreo a la cámara RGB, etiquetar manualmente las esquinas de los objetos en coordenadas de imagen, $\mathbf{X}_C \in \mathbb{R}^4$

2. Se calculan los puntos de cada objeto en coordenadas del objeto en un único *frame* de la secuencia de imágenes, ya que estas coordenadas van a ser invariables para toda la secuencia de imágenes. Para esto se deben seguir los siguientes pasos para cada objeto.

   a) Obtener del *groundtruth* la pose de la cámara respecto del mundo, $\mathbf{T}_{WC}$ y la pose del objeto respecto del mundo, $\mathbf{T}_{WO}$.

   La pose se representa con una matriz $\mathbf{T} \in \mathrm{SE}(3)$ que almacena la matriz de rotación y el vector de traslación

   $$\mathbf{T} = \begin{pmatrix} \mathbf{R} \in \mathrm{SO}(3) & \mathbf{t} \in \mathbb{R}^{3\times1} \\ 0^{1\times3} & 1 \end{pmatrix} \tag{3.1}$$

   b) Transformar los puntos del objeto en coordenadas de imagen a coordenadas del mundo, $\mathbf{X}_W \in \mathbb{R}^4$.

   $$\mathbf{X}_W = \mathbf{T}_{WC}\mathbf{X}_C \tag{3.2}$$

   c) Una vez obtenidas las coordenadas del objeto respecto del mundo, $\mathbf{X}_W$, se pasa a calcular finalmente las coordenadas del objeto respecto de este, $\mathbf{X}_O \in \mathbb{R}^4$.

   $$\mathbf{X}_O = \mathbf{T}_{WO}^{-1}\mathbf{X}_W \tag{3.3}$$

3. Tras calcular los puntos de los objetos en coordenadas del objeto en un *frame* específico, se pasa a realizar el proceso inverso para así poder obtener las coordenadas de la imagen en próximos *frames*.

   a) Obtener del *groundtruth* la pose de la cámara respecto del mundo, $\mathbf{T}_{WC}$ y la pose del objeto respecto del mundo, $\mathbf{T}_{WO}$.

   b) Transformar los puntos del objeto en coordenadas del objeto a coordenadas del mundo, $\mathbf{X}_W$.

   $$\mathbf{X}_W = \mathbf{T}_{WO}\mathbf{X}_O \tag{3.4}$$

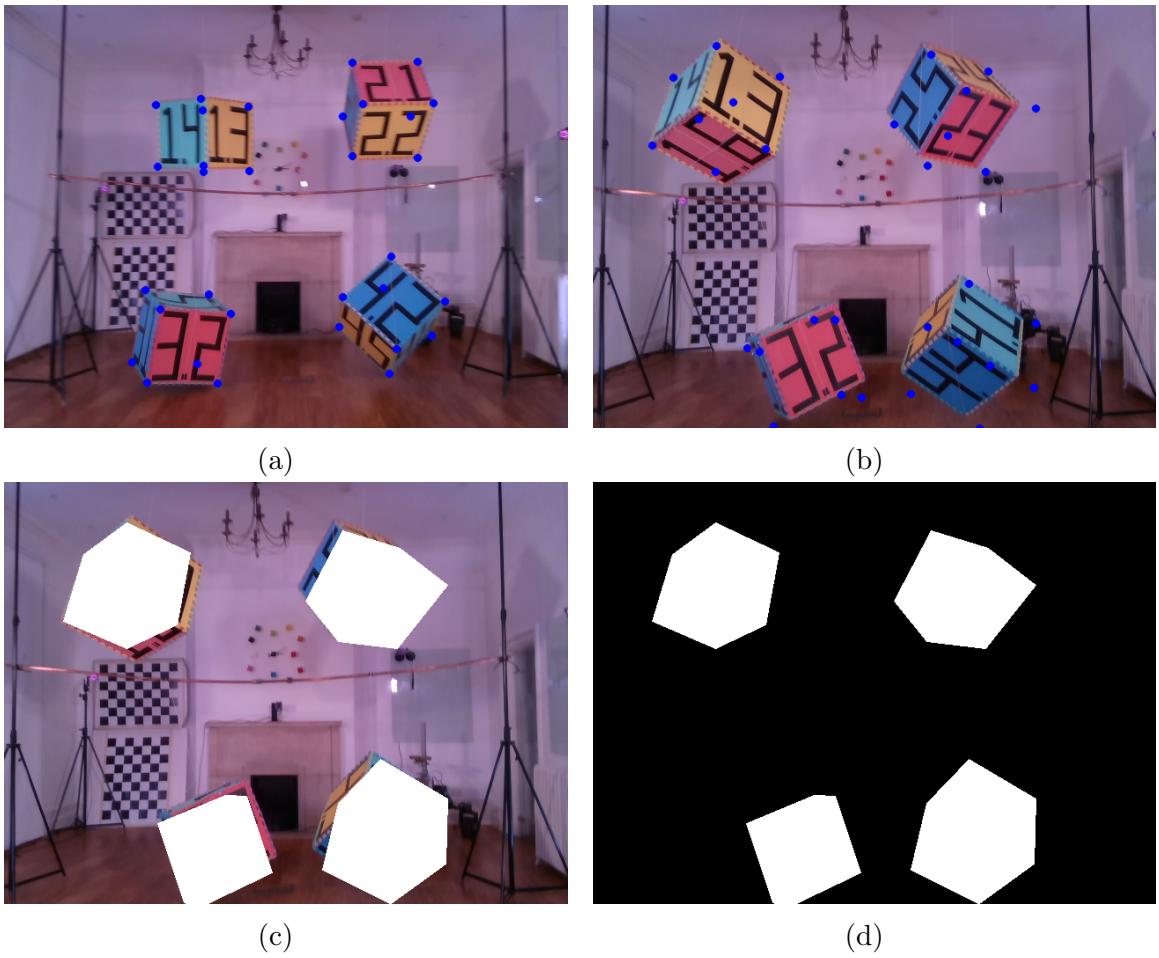
c) Una vez obtenidas las coordenadas del objeto respecto del mundo, $\mathbf{X}_W$, se pasa a calcular las coordenadas del objeto respecto de la cámara, $\mathbf{X}_C$.

$$\mathbf{X}_C = \mathbf{T}_{WC}^{-1}\mathbf{X}_W \tag{3.5}$$

d) Tras esto, se normaliza el valor de $\mathbf{X}_C$ respecto del eje Z y se calculan las coordenadas de los puntos en la imagen, $\mathbf{x} = [u, v, 1] \in \mathbb{R}^3$.

$$\mathbf{x} = \mathbf{K}(\mathbf{X}_C / \mathbf{X}_C^{(3)}) \tag{3.6}$$

4. Finalmente se calculan las máscaras uniendo los puntos entre sí (Figura 3.4).



(a) (b)

(c) (d)

Figura 3.4:
(a) Puntos introducidos para el primer *frame*.
(b) Puntos calculados para el *frame* 100.
(c) Máscara generada por la unión de los puntos.
(d) Máscara que será utilizada para calcular Precisión y Exhaustividad

El código utilizado para la implementación se encuentra en la Subsección A.0.2 del anexo.

3.4. Métricas

Una vez obtenidos *outliers*, *inliers*, máscaras y la matriz fundamental se pueden obtener diversas métricas con las que se quiere analizar la distribución de inliers y outliers en función de su pertenencia a objetos dinámicos o estáticos. A continuación se van a exponer como se han calculado las diversas métricas y los resultados que se han obtenido.

3.4.1. Error de la pose relativa

Para obtener la pose relativa estimada dados dos *frames* se ha seguido el siguiente procedimiento

1. Tras estimar la matriz fundamental, $\mathbf{F} \in \mathbb{R}^{3\times3}$, se obtiene la matriz esencial, $\mathbf{E} \in \mathbb{R}^{3\times3}$.

$$\mathbf{E} = \mathbf{K}^T\mathbf{F}\mathbf{K} \tag{3.7}$$

2. Teniendo en cuenta que $\mathbf{E} = \mathbf{R}[\mathbf{t}]_\times$, donde $\mathbf{R} \in \mathrm{SO}(3)$ es la rotación modelada por $\mathbf{E}$ y $[\mathbf{t}]_\times$ es la representación matricial del producto vectorial con el vector $\mathbf{t} \in \mathbb{R}^3$ que modela la traslación, se pueden recuperar estos valores dado $\mathbf{E}$.

   a) Se aplica la descomposición en valores singulares (SVD) a la $\mathbf{E}$

$$\mathbf{E} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{3.8}$$

   b) Una vez descompuesta $\mathbf{E}$, se pasan a calcular $\mathbf{R}$ y $\mathbf{t}$.

$$[\mathbf{t}]_\times = \mathbf{U}\mathbf{W}\mathbf{\Sigma}\mathbf{U}^T \tag{3.9}$$

$$[\mathbf{t}]_\times \overset{\text{def}}{=} \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix}. \tag{3.10}$$

$$\mathbf{R} = \mathbf{U}\mathbf{W}^{-1}\mathbf{V}^T \tag{3.11}$$

3. Después de obtener $\mathbf{R}$ y $\mathbf{t}$ se pasa a reconstruir la pose, para esto es necesario comprobar que combinación de $\mathbf{R}$ y $\mathbf{t}$ devuelven una pose físicamente posible ya que con los valores de $\mathbf{R}$ y $\mathbf{t}$ se pueden construir cuatro poses distintas.

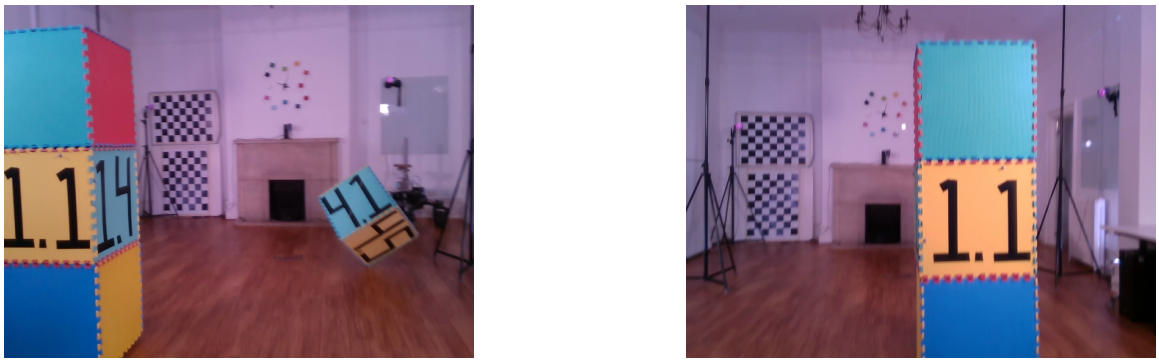
Finalmente, se multiplica la inversa de la pose estimada por la obtenida en el *groundtruth*. Esto nos devuelve como resultado la matriz identidad si la pose estimada no tiene error, en caso contrario nos devolverá el movimiento a realizar para pasar de la pose estimada a la real.
El código utilizado se encuentra en la Subsección A.0.1 del anexo.

3.4.2. Resultados

Los resultados presentados se han dividido en dos secciones diferentes correspondientes a las dos secuencias utilizadas.

occlusion_2_unconstrained

Este *dataset* nos muestra 2 objetos (Figura 3.5a), el objeto que se encuentra en primer plano se va moviendo por el suelo mientras que el segundo objeto está colgado del techo. La intuición nos dice que los movimientos del primer objeto influirán más en los errores calculados ya que este ocluye una mayor parte de la imagen, además, el segundo objeto se ve obstruido por el primero (Figura 3.5b) en numerosas ocasiones por lo que su impacto en los resultados será reducido.



(a) Ambos objetos aparecen. (b) El objeto 1 ocluye al objeto 2.

Figura 3.5: Oclusión de objetos.

En la Figura 3.6 aparecen gráficas de la precisión y exhaustividad obtenidas por el sistema tradicional en función de la distancia recorrida por cada objeto. En este sistema, la precisión representa el porcentaje del total de puntos de interés clasificados que el sistema ha clasificado como pertenecientes a objetos dinámicos. Por consecuencia, una precisión de 1 implica que si el sistema ha clasificado un punto de interés como perteneciente a objeto dinámico, podemos estar seguros de que ese objeto es dinámico. Por otra parte, la exhaustividad representa el porcentaje del total de puntos de interés que se conoce que pertenecen a objetos dinámicos es capaz de clasificar el sistema como pertenecientes a objetos dinámicos. Una exhaustividad de 1 implica que el sistema ha detectado todos los puntos pertenecientes a objetos dinámicos correctamente. Tener una precisión y exhaustividad de 1 implica que el sistema ha clasificado todos los posibles objetos dinámicos sin un único error, esto es lo que se busca para este sistema.

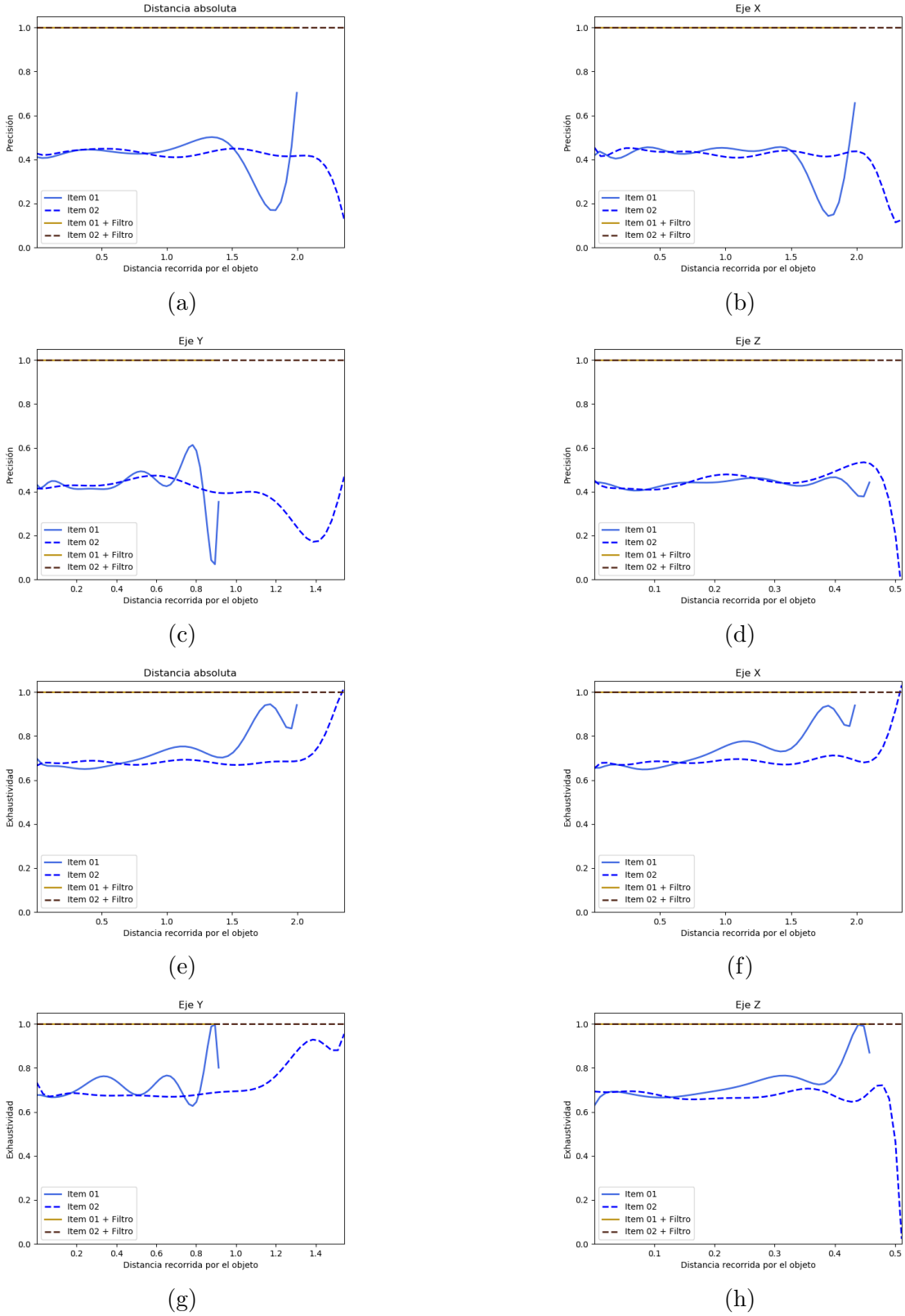


Figura 3.6: (a-d) Precisión en función de la distancia absoluta, la distancia en el eje X, la distancia en el eje Y y la distancia en el eje Z recorrida por los objetos, respectivamente. (e-h) Exhaustividad en función de la distancia absoluta, la distancia en el eje X, la distancia en el eje Y y la distancia en el eje Z recorrida por los objetos, respectivamente.

En cuanto a los resultados obtenidos por el sistema tradicional de estimación de pose que no tiene en cuenta los objetos dinámicos (líneas azules) se observa que la precisión obtenida se encuentra normalmente entre un 20 y un 50 %. Es decir, de todos los *outliers* que el sistema es capaz de detectar, aproximadamente la mitad pertenece efectivamente a objetos dinámicos y la otra mitad no. Estos emparejamientos son debidos en gran parte a patrones repetitivos en las imágenes y a la presencia de ruido. Si se utilizan máscaras para filtrar los *key points* y los emparejamientos pertenecientes a objetos dinámicos la precisión puede elevarse de manera importante y llegar a ser del 100 %. En tal caso los *outliers* únicamente corresponderían a emparejamientos erróneos en las zonas estáticas de las imágenes.

En cuanto a la exhaustividad del sistema tradicional se observa que de manera más o menos homogénea se obtiene una exhaustividad del 70 %. Es decir, aproximadamente tres cuartos de las correspondencias pertenecientes a objetos dinámicos son detectadas como *outliers* del modelo de escena estática, pero el otro cuarto no lo es. Si se utilizan máscaras para filtrar los *key points* y los emparejamientos pertenecientes a objetos dinámicos la exhaustividad puede llegar a ser de hasta el 100 %. Este resultado depende únicamente de la calidad de las máscaras estimadas.

A continuación se va a analizar cómo varían los valores de precisión y exhaustividad en función de la distancia recorrida por los objetos (Figura 3.6). Se observa que por lo general los valores de precisión y exhaustividad disminuyen y aumentan respectivamente en función de la distancia recorrida. Esta variación se observa principalmente en la precisión y exhaustividad producidas por el objeto 1 ya que es el objeto que más espacio ocupa en la imagen. Cuanta más distancia recorren los objetos más difícil es encontrar emparejamientos correctos entre ellos, ya que es posible que sus descriptores no se parezcan porque las condiciones de iluminación, rotación y escala sean muy diferentes. Si el número de emparejamientos pertenecientes a objetos dinámicos es muy bajo, entonces será más fácil que el algoritmo de RANSAC clasifique estos pocos emparejamientos como *outliers*. Esto se ve que ocurre claramente en las figuras 3.6e, 3.6f y 3.6g al llegar la exhaustividad a casi un 100 %. Por lo contrario, se observa que la precisión tiende a disminuir ligeramente al aumentar la distancia que recorren los objetos. Esto se debe a que la mayoría de los *outliers* en estos casos se deben a emparejamientos erróneos de la escena estática y prácticamente nada a aquellos de los objetos dinámicos.

El problema de los sistemas tradicionales de estimación de pose ocurre cuando el número de emparejamientos en las zonas estáticas de la imagen es similar al número de emparejamientos en un objeto dinámico. En estos casos, los algoritmos tradicionales pueden llegar a concluir que la escena estática es el objeto dinámico y entonces

computar el movimiento de éste como el de la propia cámara.

Se ha querido mostrar en la Figura 3.7 la distancia recorrida por el objeto 1 (caja grande) y del objeto 2 (caja pequeña) y cómo influye ésta en el error al calcular la pose, con una distancia entre *frames* de 50. Habiendo obtenido las máscaras de los objetos dinámicos, también se va a calcular el error de la pose estimada omitiendo los puntos de interés que se vean ocluidos por las máscaras. Destacar que las gráficas muestran los resultados de aplicar regresión polinómica a los datos obtenidos para una mayor claridad en la curva. También cabe destacar que la pose estimada no tiene información sobre la escala de la escena debido a que se está trabajando con un sistema monocular. Por lo tanto los resultados mostrados en este apartado carecen de escala y únicamente tienen sentido si se comparan con otros resultados similares en la misma escala.

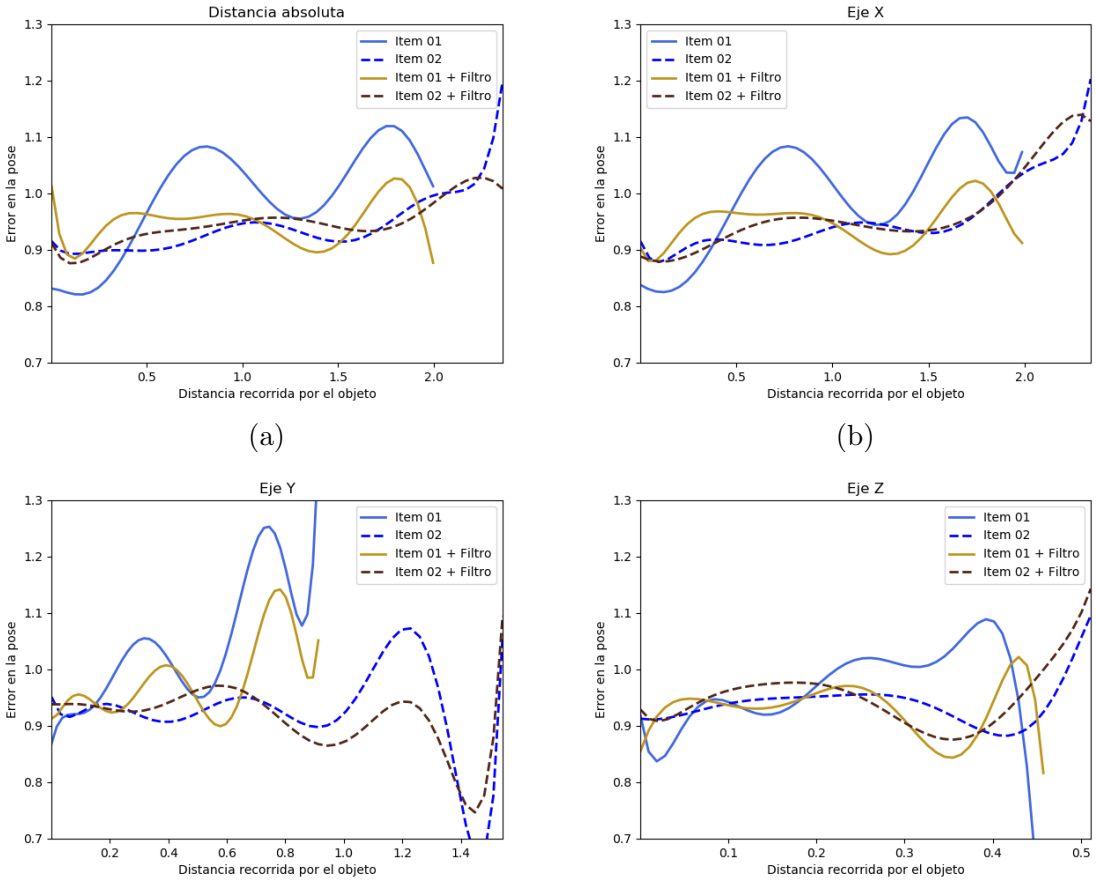


Figura 3.7:

(a-d) Error en la pose en función de la distancia absoluta , la distancia en el eje X, la distancia en el eje Y y la distancia en el eje Z de la cámara recorrida por los objetos, respectivamente.

En la Figura 3.7a se puede observar el error de la pose *up to scale* en función de la distancia recorrida por los objetos. Se muestra el error con el algoritmo tradicional de estimación de pose con las líneas azules, y con líneas amarillas los resultados filtrando previamente aquellos emparejamientos coincidentes con las máscaras de los objetos dinámicos. Principalmente en el caso del objeto 1, que es el más representativo, se observa una mejora considerable en el error de la pose.

Hemos considerado interesante cómo afecta el descomponer el movimiento de los objetos en los 3 ejes principales de la cámara. El movimiento en la coordenada X es el movimiento que más influye en los errores al calcular la pose, esto se debe a que la mayor parte del movimiento se realiza en este eje. Finalmente se observa que el movimiento en el eje Z influye mucho menos en el error obtenido.

swinging_4_unconstrained

Este *dataset* nos muestra 4 objetos (Figura 3.8) balanceándose en diferentes ejes, al contrario que en *dataset* anterior, no hay oclusión de la imagen en ningún momento. El movimiento se realiza principalmente en los ejes Xy Z.

Figura 3.8: Todos los objetos que aparecen en este *dataset*.

Los resultados obtenidos para este *dataset* comparten muchas similitudes con los obtenidos para el *dataset* anterior siendo la más notable la disminución del error al aplicar las máscaras, por lo que en este apartado solo se comentarán aquellas cosas que se consideren destacables.

En cuanto a precisión y exhaustividad (Figura 3.9), se observa que influencia de la distancia recorrida por los objetos es mucho menor si se compara con la observada en la Figura 3.6. Esto es debido a que el movimiento de los objetos es muy reducido y se ve limitado por la longitud de las cuerdas y la fuerza aplicada a estos para el balanceo. También se puede ver que la precisión obtenida es considerablemente menor a la obtenida con el *dataset* anterior obteniéndose en torno a un 30 %. Esto se debe a la disminución en el movimiento relativo de los objetos, ya que cuanto menor sea el movimiento, menor número de puntos de interés serán determinados

pertenecientes a objetos dinámicos. La exhaustividad, en cambio, tiene valores muy similares a los comentados en la Figura 3.6.

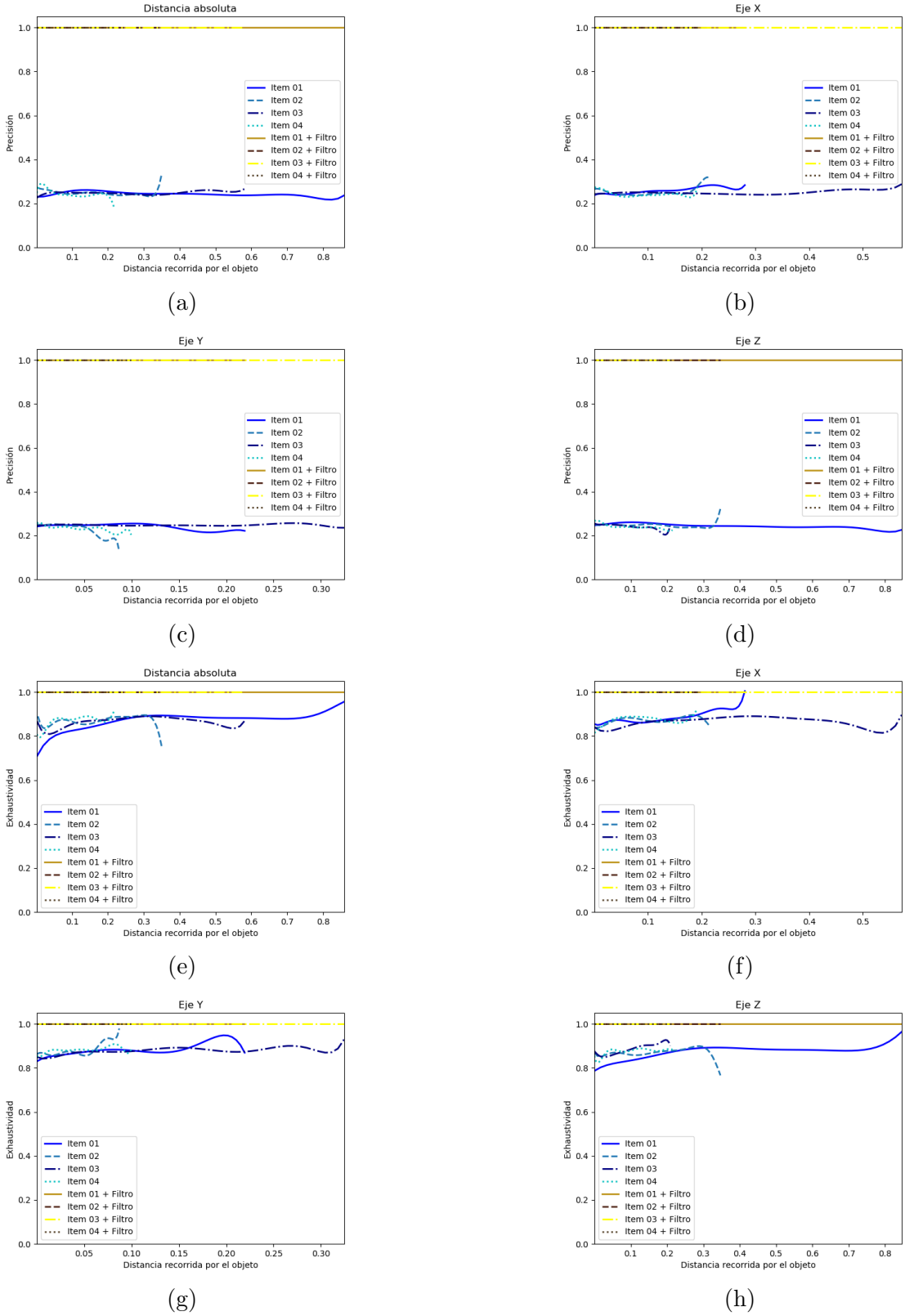


Figura 3.9: (a-d) Precisión en función de la distancia absoluta, la distancia en el eje X, la distancia en el eje Y y la distancia en el eje Z recorrida por los objetos, respectivamente. (e-h) Exhaustividad en función de la distancia absoluta, la distancia en el eje X, la distancia en el eje Y y la distancia en el eje Z recorrida por los objetos, respectivamente.

En la Figura 3.10 se observa, como ya se ha comentado anteriormente, el error de calculo disminuye al utilizar los filtros.

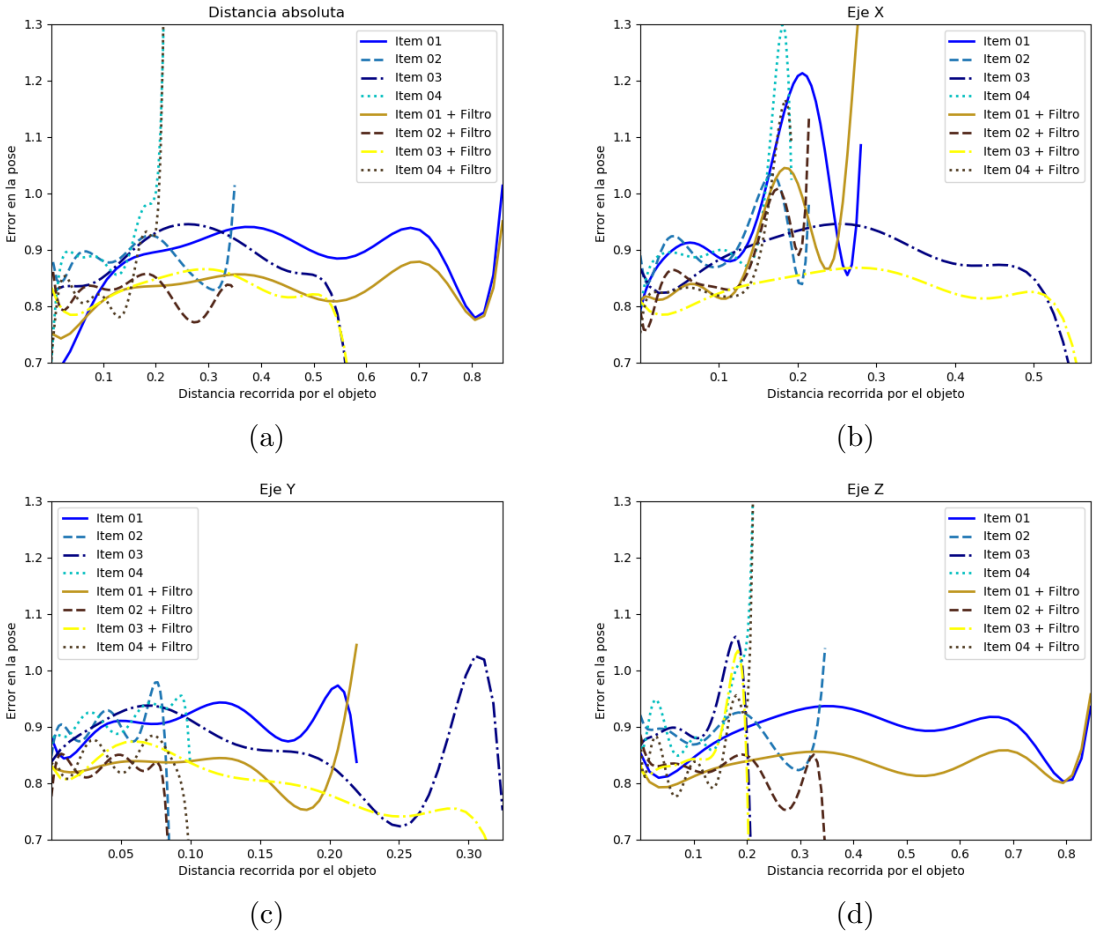


Figura 3.10:

(a-d) Error en la pose en función de la distancia absoluta, la distancia en el eje X, la distancia en el eje Y y la distancia en el eje Z recorrida por los objetos, respectivamente.

3.4.3. Conclusiones

Como se ha podido ver en las Figuras 3.7 y 3.10 la omisión de los objetos dinámicos proporciona mejoras considerables a la hora de estimar la pose de la cámara, por lo que un sistema capaz de detectar los objetos dinámicos implicaría un gran paso adelante en sistemas que basan su funcionamiento en conocer la pose de la cámara. El principal problema es que el sistema propuesto en este capítulo no es lo suficientemente robusto como para poder utilizarse en sistemas comerciales, por lo que en el Capítulo 4 se propone un sistema basado en redes neuronales convolucionales que busca clasificar los objetos dinámicos con una mayor consistencia.

Capítulo 4

Detección de Objetos Dinámicos

En el capítulo anterior se ha visto que con los métodos tradicionales de detección de *outliers* basados en RANSAC es posible conseguir una precisión entre 20 y un 50 % y una exhaustividad entre un 70 y un 80 % en la detección de *key points* pertenecientes a objetos dinámicos. Puesto que con el uso de máscaras de objetos dinámicos se puede obtener una precisión y una exhaustividad de hasta el 100 % y esto implica una estimación más precisa de la pose de la cámara, se quiere en este capítulo del TFG desarrollar una red neuronal convolucional que sea capaz de determinar de manera consistente qué objetos de la escena son dinámicos. Se quiere resaltar que también se utilizará RNC para referirse a red neuronal convolucional en este capítulo.

4.1. Planteamiento

La estructura elegida para implementar el sistema se divide en 3 módulos y se inspira en la propuesta para el articulo *Dynamic Attention-based Visual Odometry* [30]:

- Red neuronal. Implementado para este proyecto.
- Generador de flujo óptico. Implementado para este proyecto.
- Extractor de máscaras. Se ha utilizado YOLACT [24] debido a su velocidad de ejecución y la capacidad de determinar en número de máscaras que queremos que genere.
- Post-procesado. Implementado para este proyecto.

El modelo de red neuronal propuesto (Figura 4.1) toma como entrada dos imágenes de una misma escena, calcula el flujo óptico entre estas dos imágenes y pasa la primera imagen al extractor de máscaras, que generará las 100 máscaras más relevantes. Una vez obtenidas las 100 máscaras y el flujo óptico, se concatenarán para formar un tensor de 101 elementos que se pasará al modelo para que este realice sus predicciones.

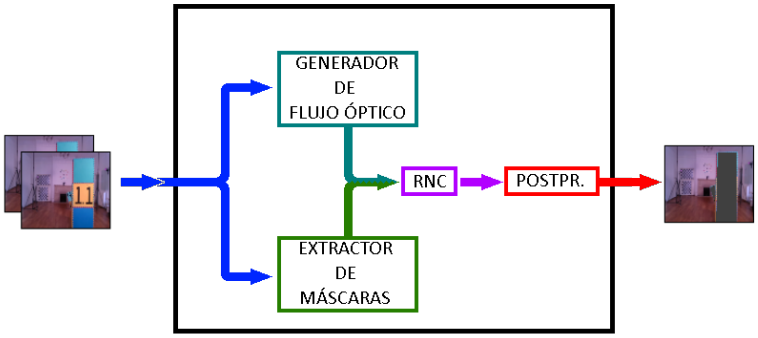


Figura 4.1: Estructura del sistema implementado.

Como resultado, el sistema generará un tensor de 100 números reales entre 0 y 1 que representarán la confianza con la que el modelo cree que esos objetos son dinámicos o no, cuanto más cercano a 1 mayor probabilidad de que este objeto sea dinámico según el modelo. Este tensor se pasará al módulo de post-procesado que generará una imagen con máscaras en los objetos dinámicos.

4.2. Modelo

En la Figura 4.2 se ilustra y describe la estructura del modelo de RNC que mejor resultados ha producido. Este modelo ha sido implementado siguiendo una estructura secuencial con 5 capas de convolución, 2 capas densas y diversos mecanismos de regularización, a continuación se van a comentar los aspectos más importantes del modelo.

- Capas de convolución. Tienen como principal objetivo reducir el total de las imágenes de entrada a un conjunto de filtros que contengan las características más importantes de cada una de las imágenes. A las capas densas se enviarán un tensor de $101 \times 5 \times 5 \times 32$.

- Capas densas. La capa densa final es la encargada de generar los resultados de la red, utiliza una función de activación sigmoidal que convierte cualquier valor de entrada a un número real dentro del intervalo [0,1]. La penúltima capa densa es donde la mayor parte del proceso de regularización recae ya que esta capa contiene más de 40 millones de parámetros entrenables.

– Mecanismos de regularización. Para evitar el sobreajuste en el entrenamiento, se han incorporado varios mecanismos de regularización, como pueden ser varias capas de *Dropout* y un regularizador L2 en las capas convolucionales y en la penúltima capa densa. El regularizador L2 es un método de regularización basado en penalizar aquellos modelos más complejos; un modelo es denominado "simple" cuando la distribución de los valores de sus parámetros tiene menos entropía. Por lo que para mitigar el sobreajuste se obliga a los pesos del modelo a tomar valores más pequeños regularizándose así la distribución de estos.

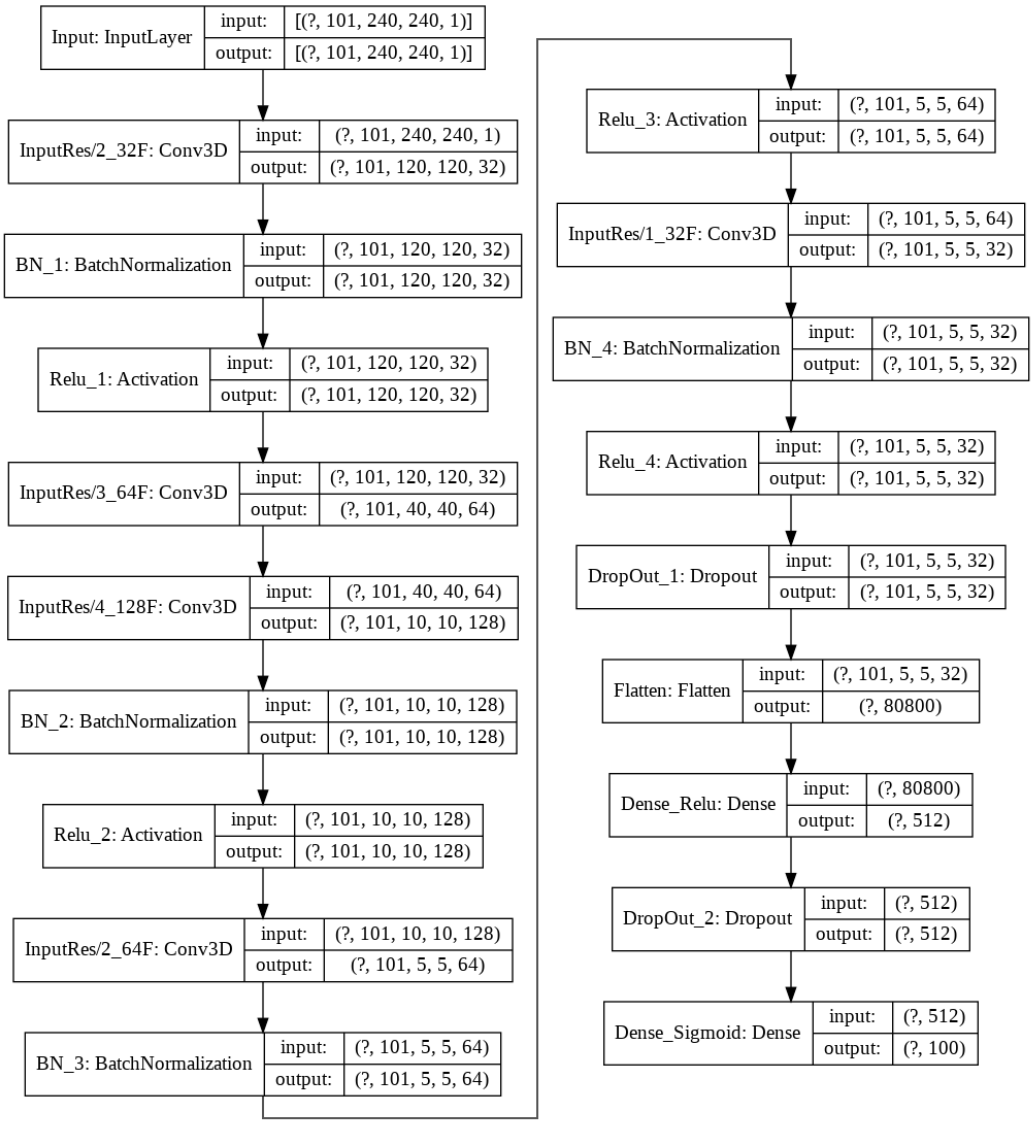


Figura 4.2: Modelo implementado. Donde "?" representa el tamaño del *batch* que es desconocido hasta que comienza la ejecución.

4.3. Máscaras

Para el extractor de máscaras es necesario el uso de una red de segmentación semántica capaz de diferenciar los objetos que aparecen en una imagen y superponer una máscara sobre estos. En un principio se consideró utilizar Mask-RCNN [23] para esta tarea, pero por diversos motivos se ha acabado utilizando YOLACT [24]. Las ventajas de YOLACT respecto de Mask-RCNN radican en los siguientes puntos.

- Velocidad de ejecución. YOLACT está diseñada para ser ejecutada en sistemas de tiempo real, mientras que los tiempos de ejecución de Mask-RCNN son lo suficientemente elevados como para no poder ser utilizados en este tipo de sistemas.
- Proceso de instalación. Para poder instalar Mask-RCNN es necesario instalar una versión de TensorFlow y de CUDA obsoletas, aunque hay esfuerzos por parte de la comunidad de actualizar el código a versiones más recientes. En cambio, YOLACT está desarrollado para la versión más reciente de PyTorch (otra biblioteca de aprendizaje profundo) y es compatible con versiones recientes de CUDA
- Grado de personalización. YOLACT permite establecer un número mínimo de máscaras a generar para una imagen dada y son raras las ocasiones en las que no se alcanza ese número, mientras que Mask-RCNN permite establecer un límite superior de máscaras. Para este trabajo es necesario generar 100 máscaras (así nos aseguramos de obtener máscaras para la gran mayoría de objetos), para lo que Mask-RCNN tenia problemas, mientras que YOLACT las genera perfectamente.
- Pesos pre-entrenados. YOLACT pone a disposición del usuario una mayor variedad de pesos pre-entrenados que Mask-RCNN.
- Calidad de las máscaras para objetos desconocidos. La Figura 4.3 muestra los resultados obtenidos de ejecutar Mask-RCNN y YOLACT sobre la misma imagen, devolviendo la máscara más relevante detectada por cada modelo.

(a) Entrada. (b) Salida de Mask-RCNN. (c) Salida de YOLACT.

Figura 4.3:

(a) Imagen original.
(b) Máscara del objeto 1 extraída por Mask-RCNN.
(c) Máscara del objeto 1 extraída por YOLACT.

4.4. Flujo óptico

Para la implementación del generador de flujo óptico se han seguido los siguientes pasos.

1. Calcular puntos de interés en ambas imágenes de entrada (Figura 4.4). Para el cálculo de puntos de interés se ha utilizado el extractor implementado para ORB-SLAM [21] puesto que, comparando con el extractor de OpenCV, calcula los puntos de interés más dispersos en la imagen.

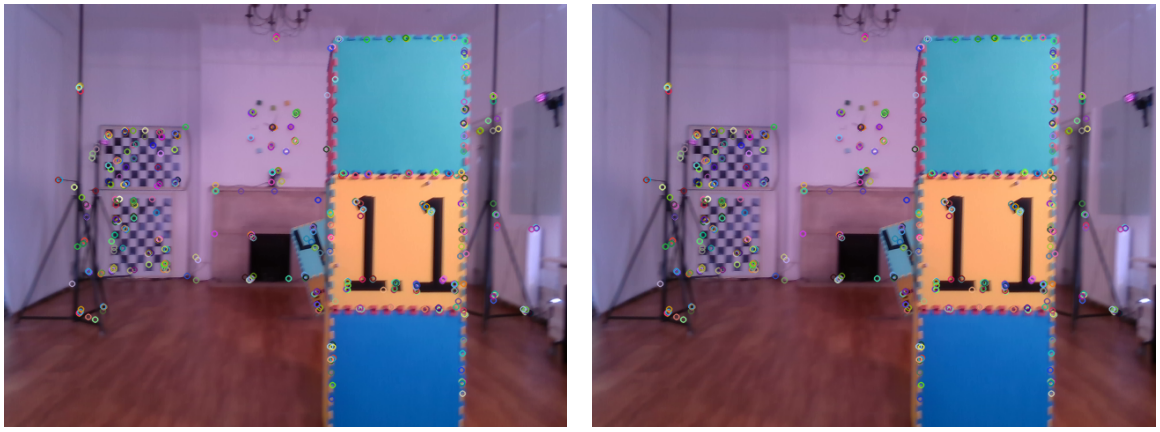


Figura 4.4: Puntos de interés extraídos.

2. Calcular emparejamientos (Figura 4.5). Tras obtener los puntos de interés se pasa al proceso de emparejamiento. El emparejamiento se ha realizado mediante un algoritmo de fuerza bruta, a su resultado se ha aplicado el ratio del segundo vecino para filtrar los emparejamientos más débiles.

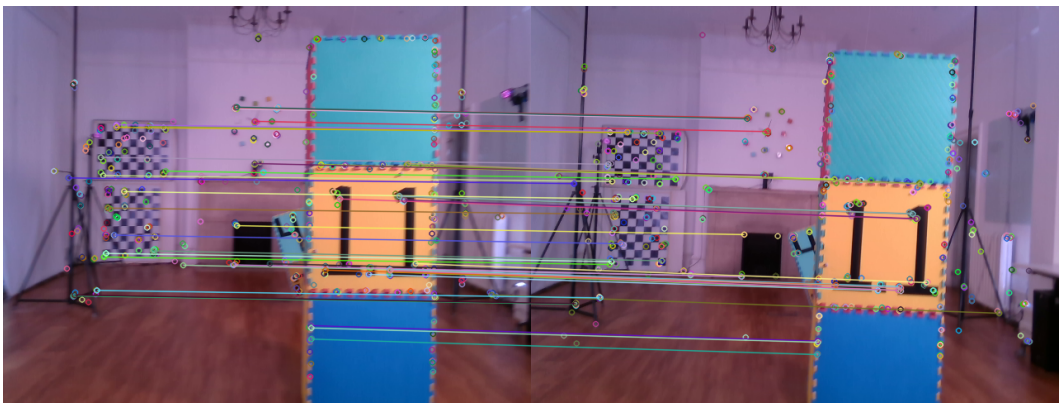


Figura 4.5: Emparejamientos realizados.

3. Filtrar emparejamientos erróneos y dibujar flujo óptico (Figura 4.6). Una vez calculados los emparejamientos se pasa a filtrar aquellos cuya separación en cuanto a distancia sea mayor a cierto valor indicado para así eliminar emparejamientos erróneos. Tras esto, se dibuja el flujo óptico en una imagen con fondo negro.

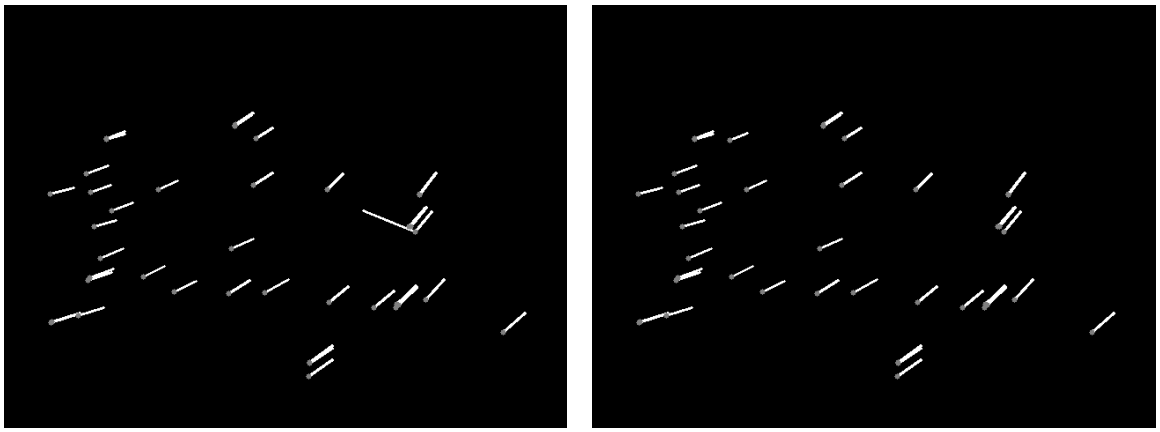


Figura 4.6: Flujo óptico antes y después del filtrado.

4.5. Post-procesado

Los resultados generados por la RNC son valores reales entre 0 y 1 y en raras ocasiones serán únicamente 0 o 1, por lo que es necesario definir un umbral a partir del cual considerar un valor como 0 o como 1. El módulo de post-procesado se encarga de esto, además de dibujar sobre la imagen inicial aquellas máscaras que superen ese umbral.

4.6. *Dataset* utilizado

Se ha utilizado el *dataset* de Oxford [27], el mismo que se ha utilizado a lo largo del proyecto, esto es debido a que ya se tienen las máscaras correspondientes a los objetos dinámicos. Para generar los datos con los que entrenará la red neuronal se han seguido los siguientes pasos

- Generar flujos ópticos. El flujo óptico utilizado para el entrenamiento se ha generado utilizando el extractor de puntos de interés de ORB-SLAM [21], detectando un total de 4000 puntos de interés en cada imagen. Las imágenes están separadas entre sí por 5 *frames*.

- Extraer máscaras. Las máscaras se han generado mediante YOLACT [24], generándose un total de 100 máscaras para cada *frame*.

- Proceso de etiquetado. Se ha realizado comparando las máscaras devueltas por YOLACT con las máscaras generadas para ambos *datasets* de Oxford (Proceso explicado en Sección 3.3), se considera que una máscara devuelta por YOLACT es de un objeto dinámico cuando el resultado de calcular la intersección sobre la unión con la máscara del *dataset* sea superior a 0.8.

4.7. Entrenamiento

Al comienzo del desarrollo de la red, el entrenamiento se realizó enteramente en un ordenador portátil Acer Aspire A515-51G, con un procesador Intel i5-8250U, 8GB de memoria RAM, tarjeta gráfica NVIDIA GeForce MX130 y en el sistema operativo Linux Mint. Se tuvieron problemas a la hora de configurar CUDA así que el entrenamiento se realizó enteramente en CPU. Tras establecer las bases de la red, se pasó a afinar el proceso de entrenamiento utilizando Google Colab que proporciona entornos de ejecución configurados para entrenar modelos con aceleración GPU. Una vez se asentaron los hiperpárametros, se contrató el servicio PRO que ofrece Google Colab. El servicio PRO nos proporciona acceso a GPUs más potentes además de la eliminación del limite de tiempo de ejecución, por lo que finalmente el proceso de entrenamiento se realizó usando esta plataforma. De todos modos, el tiempo necesario para acabar el entrenamiento es muy elevado, aunque notablemente menor que cuando se entrenaba en el ordenador portátil.

El modelo presentado en la Sección 4.2 se ha entrenado con los siguientes hiperparámetros:

- Optimizador. El optimizador con el que mejores resultados se ha conseguido ha sido Adam con un ratio de aprendizaje de 0.00005 consiguiéndose una convergencia alrededor de la decimoquinta época mientras que para el resto de optimizadores era necesario entrenar durante más de 30 épocas para alcanzar unos resultados decentes. El ratio de aprendizaje óptimo se ha obtenido utilizando un algoritmo de búsqueda de ratio de aprendizaje (*LR finder*) como el comentado en el articulo *Cyclical Learning Rates for Training Neural Networks* [31], este algoritmo aumenta periódicamente el valor del ratio de aprendizaje durante una época del entrenamiento almacenando los costes obtenidos para cada uno de estos valores, tras la ejecución se muestra una gráfica que representa el ratio de aprendizaje en el eje X y el coste en el eje Y. El ratio de aprendizaje óptimo será aquel a partir del cual hay mayor pendiente en el coste (Figura 4.7).

- Función de coste. La función de coste utilizada es *Binary crossentropy* o entropía cruzada binaria. La entropía cruzada es una función de coste que mide el rendimiento de un modelo cuya salida es una probabilidad entre 0 y 1, el coste devuelto crece conforme la probabilidad predicha diverge del resultado esperado. La entropía cruzada binaria sigue el mismo principio pero los resultados esperados son 0 o 1.

- Número de épocas. El modelo empieza a dar buenos resultados a partir de 15 épocas de entrenamiento pero el proceso se ha alargado hasta un total de 50 épocas para garantizar la convergencia del coste.

- Tamaño del *dataset*. El entrenamiento se ha realizado con un total de 10000 datos, de los cuales 7000 se han usado para entrenamiento, 2000 para validación y los 1000 restantes para test.

- Cálculo de la precisión. Para calcular la precisión del modelo se ha utilizado la métrica de *binary accuracy*, puesto que el resultado esperado es un vector de valores binarios.

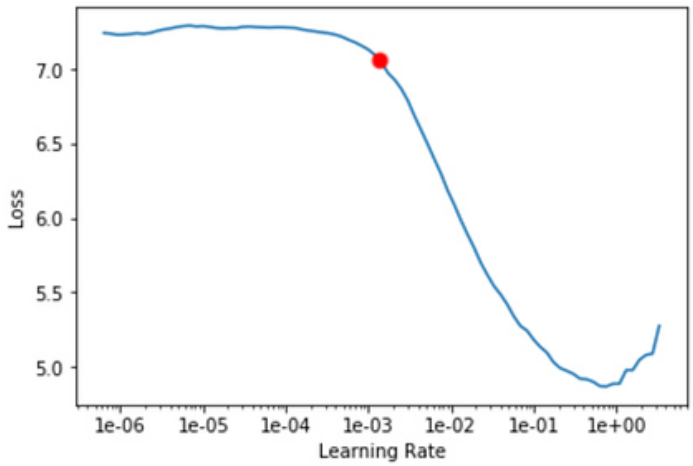


Figura 4.7: Ejemplo de gráfica generada por un algoritmo de búsqueda de ratio de aprendizaje, en rojo marcado el valor óptimo.

4.8. Resultados

Con la configuración comentada en el apartado anterior, se ha alcanzado una precisión superior al 90 % para los datos de validación (Figura 4.8).
Cabe destacar que la red neuronal, por lo general devuelve valores reales entre 0 y 1, por lo que es necesario definir un valor que sea un umbral para estos valores. En los resultados mostrados el umbral tendrá valor por defecto de 0.5.

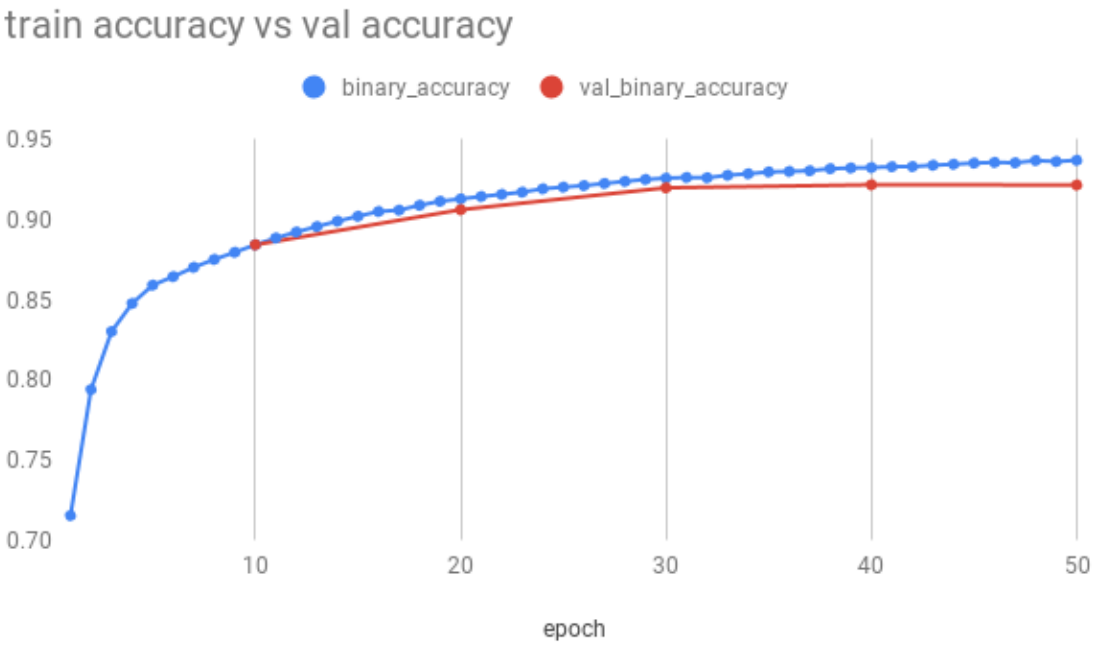


Figura 4.8: Evolución de la precisión del modelo durante el entrenamiento.

La Figura 4.9 muestra el resultado de aplicar la red neuronal implementada a dos imágenes extraídas de la secuencia de vídeo 0004 del *dataset* de KITTI [4]. En esta escena se puede observar que el coche que circula en dirección contraria se mueve mientras que el otro coche que aparece en la escena está detenido, esto se ve reflejado en el resultado ya que la única máscara que aparece en este es la que ocluye el coche de la izquierda. Queremos destacar que el hecho de que nuestra red neuronal únicamente sea entrenada con imágenes binarias en vez de con imágenes RGB hace que sus pesos generalicen a todo tipo de escenas. Esto nos permite entrenar nuestro modelo con imágenes del *Oxford Multimotion Dataset* y utilizarlo en un *dataset* completamente diferente como es el KITTI y obtener resultados satisfactorios.

Figura 4.9: Resultado de aplicar la red neuronal a la secuencia 0004 de KITTI [4].

La Figura 4.10 y la Figura 4.11 tienen unas condiciones muy similares a la Figura 4.9 ya que aparece un objeto dinámico entre varios que no lo son, como se puede ver los resultados son satisfactorios. Finalmente, en la Figura 4.12 se muestran dos imágenes de una secuencia de vídeo en la que varios coches circulan por una carretera en ambos sentidos, como se puede observar en el resultado la red es capaz de determinar correctamente como dinámicos todos los coches.

Figura 4.10: Resultado de aplicar la red neuronal a la secuencia 0007 de KITTI [4].

Figura 4.11: Resultado de aplicar la red neuronal a la secuencia 0009 de KITTI [4].

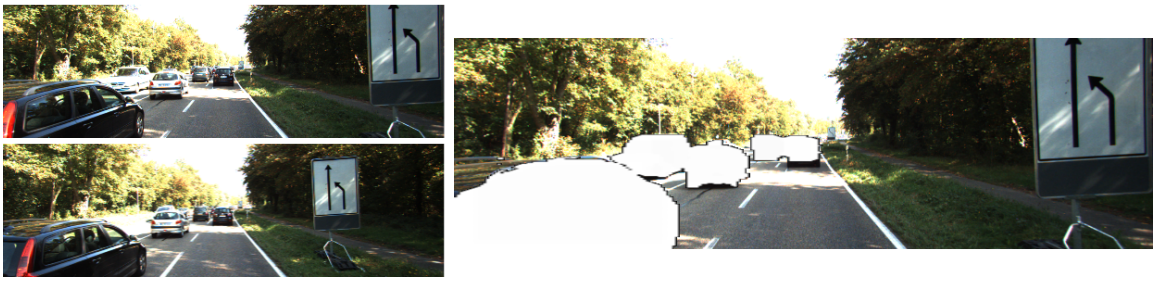
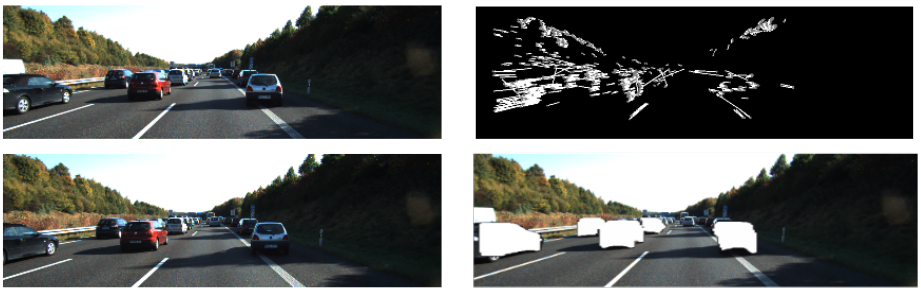


Figura 4.12: Resultado de aplicar la red neuronal a la secuencia 0018 de KITTI [4].

Como punto de partida estos resultados son muy prometedores. Sin embargo no se obtienen buenos resultados siempre.

(a)



(b)

Figura 4.13: Resultado de aplicar la red neuronal a las secuencias 0004 y 0020 de KITTI [4], respectivamente.

En la Figura 4.13 se muestran los resultados obtenidos en diversos entornos con algunos errores en la detección de objetos dinámicos. En la Figura 4.13a se debe a que el módulo de flujo óptico no ha sido capaz de hacer el proceso de emparejamiento correcto. Si se observa la zona central de la Figura 4.13b, se puede ver que no se han enmascarado los coches más lejanos, esto es debido a una limitación del sistema más que a un error. Esta limitación tiene que ver principalmente con la resolución de las imágenes de entrada que admite la red, la cual acepta únicamente imágenes de tamaño 240×240, por lo que al reducir la imagen del flujo óptico se pierde información relativa a los objetos del fondo (Figura 4.14). Esta perdida de información de los objetos del fondo se debe a que estos objetos se encuentran en el punto de fuga de la escena, por lo que al moverse hacia el frente, el tamaño relativo de estos varía muy poco.

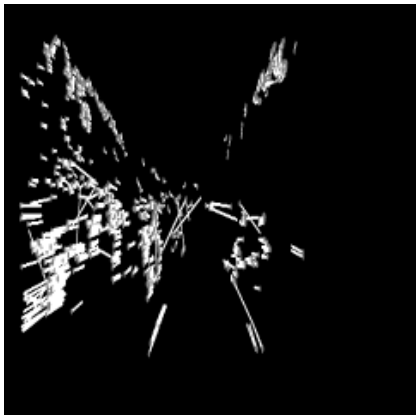
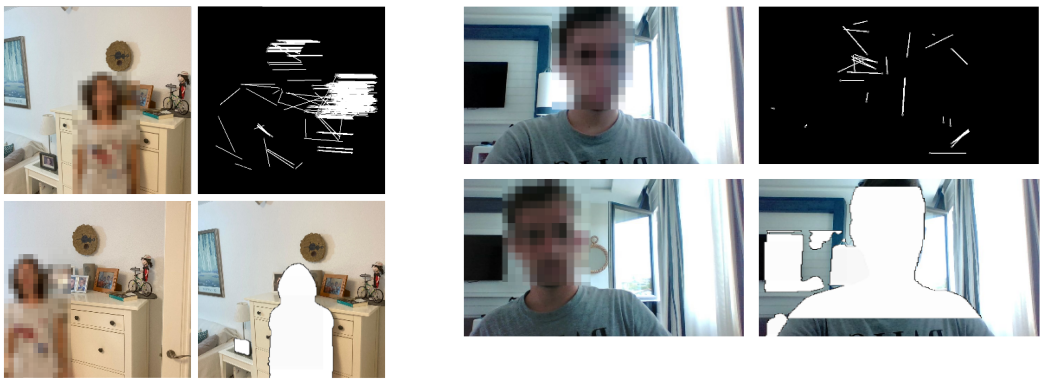


Figura 4.14: Resultado de reducir la imagen de flujo óptico

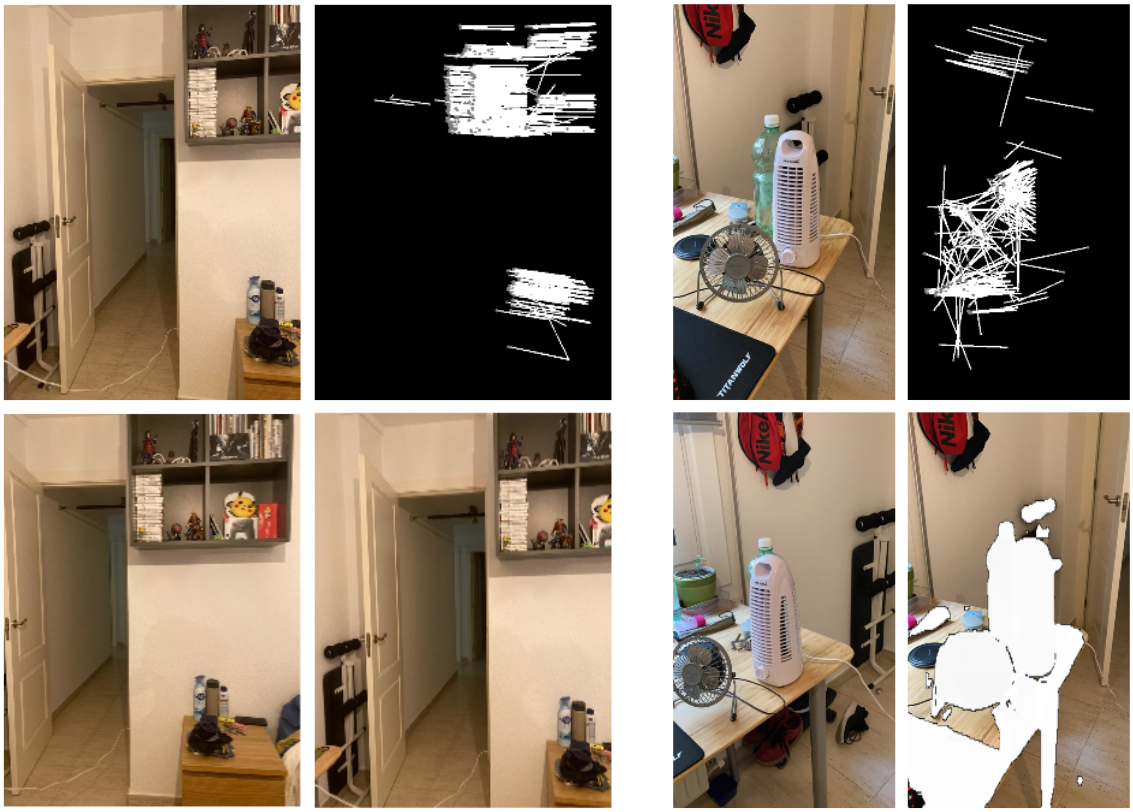
En la Figura 4.15 se muestra como la red neuronal detecta de la manera esperada el movimiento de objetos, aunque también se seleccionan como dinámicos algunos elementos incorrectos.



(a) (b)

Figura 4.15: Resultado de aplicar la red neuronal a imágenes con personas que se mueven.

En la Figura 4.15a se observa que el sistema enmascara un marco de fotos como objeto dinámico, esto puede deberse a cómo se ha entrenado la red. En el *dataset* utilizado cuando en un *frame* aparece un objeto y al siguiente no, suele ser debido a que el objeto nuevo es un objeto dinámico Figura 3.5a y Figura 3.5b, por lo que la red considera que el marco de fotos es un objeto dinámico. En la Figura 4.15b se observa que aparecen muchos artefactos, esto es debido a la calidad del flujo óptico calculado ya que esta red se ha diseñado con la idea de que el flujo óptico de entrada sea considerablemente más denso (similar al que aparece en la Figura 4.15a). El hecho de que estas imágenes tengan una textura un poco pobre hace que el flujo óptico no sea lo suficientemente informativo para detectar objetos dinámicos.



(a) (b)

Figura 4.16: Resultado de aplicar la red neuronal a imágenes sin objetos dinámicos.

Finalmente, en la Figura 4.16, se muestran resultados a destacar. En la Figura 4.16a se observa que, si las dos imágenes muestran una escena desde dos perspectivas distintas donde no aparecen objetos dinámicos, y si el flujo óptico es mínimamente bueno, la red considerará de forma correcta que ningún objeto es dinámico. En cambio, como se observa en la Figura 4.16b que muestra la misma escena desde dos perspectivas distintas, el módulo de flujo óptico tiene muchos errores a la hora de hacer el emparejamiento y genera un flujo óptico malo, lo que implica malos resultados; estos

errores son debidos a que las imágenes de entrada muestran la misma escena desde dos puntos de vista muy distintos que confunden al emparejador de puntos de interés.

También se ha calculado la curva de precisión-exhaustividad media de ejecutar el sistema sobre una secuencia de 50 imágenes extraídas del *dataset* KITTI [4], concretamente de la secuencia número 20 (la misma que se ha utilizado en la Figura 4.13b). Para esto, se ha calculado *frame* a *frame* la precisión y exhaustividad resultante de aplicar el sistema a 2 imágenes separadas por 5 *frames* de diferencia, e ir variando el umbral de clasificación desde 0 hasta 1. Finalmente se ha hecho la media entre todos los resultados. Para determinar si una máscara se había clasificado correctamente, se han determinado manualmente que máscaras generadas por YOLACT son correctas para cada *frame*.

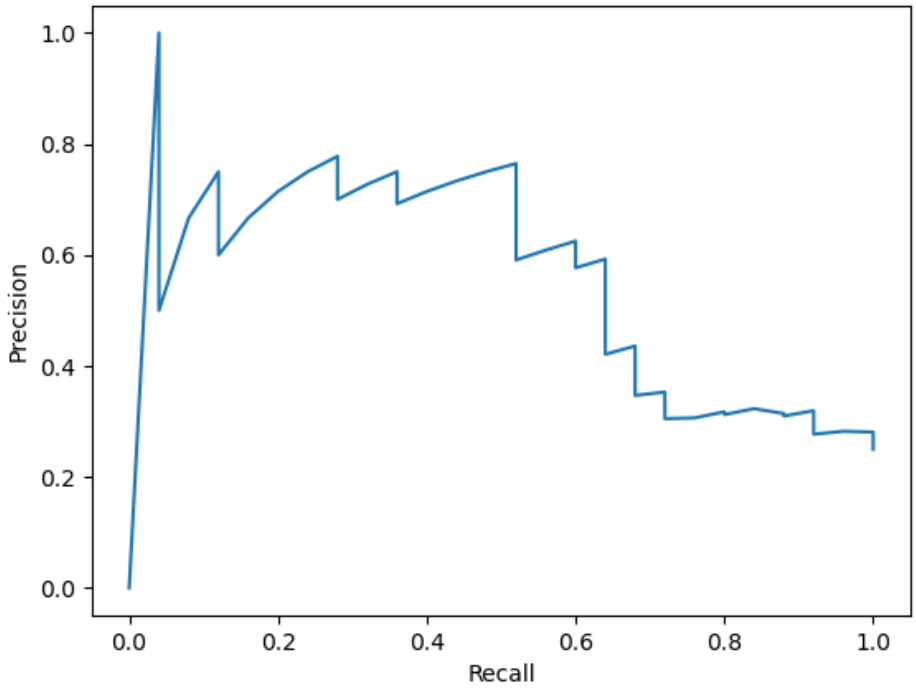


Figura 4.17: Curva Precisión-exhaustividad.

En la curva precisión-exhaustividad de la Figura 4.17 se observa que el área bajo la curva aún no es todo lo grande que podría ser, se espera aumentar este área con las mejoras que se proponen en el próximo apartado.

4.9. Posibles mejoras

Debido al poco tiempo de desarrollo del modelo y a pesar de haber obtenido buenos resultados, no se ha conseguido el rendimiento deseado, por lo que a continuación se presentan posibles mejoras a realizar para alcanzar mayor precisión en las predicciones.

- Mejorar las etiquetas usadas para el entrenamiento. Actualmente las etiquetas utilizadas para el entrenamiento se generan mediante el proceso explicado en Sección 4.6, una mejora obvia sería que estas etiquetas fueran generadas por un ser humano pero esto podría conllevar demasiadas horas de trabajo así que la opción más sensata sería mejorar el algoritmo estableciendo unas condiciones más robustas frente a valores espurios.

- Optimizar código de detección. La implementación del sistema representado en la Figura 4.1 realizada para este proyecto es muy poco eficiente. Esto se debe a que se ha implementado en Python y para calcular el flujo óptico hay que ejecutar un programa escrito en c++, puesto que se utiliza el extractor de puntos de interés de ORB-SLAM [21] que está implementado en c++ y no se ha tenido el suficiente tiempo como para traducir el código a python. Aunque el principal motivo por el cual el programa es poco eficiente es el uso de YOLACT [24] como extractor de máscaras; YOLACT ha sido implementado con el *framework* de aprendizaje profundo PyTorch, esto implica que cada vez que se quieran extraer las máscaras de una imagen es necesario ejecutar la secuencia de inicialización de PyTorch. En este proceso se dedica una gran cantidad de tiempo de ejecución del programa malgastada en cargar bibliotecas. Esto podría solventarse realizando una implementación de YOLACT para TensorFlow o simplemente optimizando el código implementado para invocar a YOLACT.

- Mejorar el generador de flujo óptico. Tal y como está implementado el sistema en este momento, los resultados obtenidos dependen enormemente de la calidad del flujo óptico obtenido. Se plantea re-entrenar la red utilizando flujo óptico denso en lugar de la aproximación realizada. El flujo óptico denso calcula el vector de flujo óptico para todos los píxeles de la imagen, esto implica mayor tiempo de cálculo para obtener mejores resultados. Se podría utilizar una red neuronal como la descrita en *FlowNet: Learning Optical Flow With Convolutional Networks* [32] para estimar el flujo óptico.

– Aumentar tamaño del *dataset*. Tal y como se ha entrenado la red neuronal los datos de entrenamiento provienen únicamente del *dataset* de Oxford [27], no vendría mal utilizar un *dataset* con mayor variedad de escenarios para así poder entrenar en casos conflictivos como el representado en la Figura 4.15a.

– Secuencias de vídeo. Tal y como está implementado en este momento solo se tienen en cuenta 2 imágenes para generar el resultado, en casos en los que las imágenes de entrada sean parte de una secuencia de vídeo, los resultados de *frames* anteriores podrían tener peso sobre el resultado del *frame* actual consiguiéndose así mayor consistencia de resultados durante el vídeo.

El código utilizado para implementar esta red neuronal se encuentra en la Subsección A.0.4 del anexo.

Capítulo 5

Problemas

En este apartado se comentan los problemas a los que me he enfrentado a lo largo del proyecto.

El dataset utilizado para la primera parte del proyecto contiene un patrón de tablero de ajedrez en la escena, este patrón provoca una concentración de puntos de interés en una zona muy pequeña lo que provoca un gran número de errores a la hora de hacer el emparejamiento. Solventar esto no fue trivial en su momento, fue necesario que Berta me recomendara utilizar el extractor de puntos de interés de ORB-SLAM [21] y aun así no se conseguían buenos emparejamientos siempre. Finalmente se implementó un sistema de emparejamiento más robusto que funciona conjuntamente con el extractor de puntos de interés comentado anteriormente.

Inicialmente, se planteó hacer uso de Mask-RCNN para la red neuronal, pero al final se decidió hacer uso de YOLACT. Para instalar Mask-RCNN y que funcione correctamente la aceleración por GPU es necesario instalar una versión de CUDA compatible con TensorFlow 1.8, esto implica instalar una versión de CUDA obsoleta para una versión de tensorflow obsoleta. Durante la instalación de CUDA surgieron diversos errores de compatibilidad de drivers, para solventar estos errores es necesario hacer una instalación limpia de todos los drivers de NVIDIA por lo que finalmente se optó por no utilizar la aceleración por GPU de Mask-RCNN.

La red neuronal se implementó en TensorFlow 2.1 y Google Colab utiliza TensorFlow 2.3 por lo que fue necesario re-implementar ciertas partes de la red. Tras esto, durante el proceso de entrenamiento las precisiones obtenidas eran muchísimo peores a las obtenidas en TensorFlow 2.1, esto es debido a que en la versión 2.3 hay que especificar de manera implícita que función de cálculo de precisión se va a utilizar, en este caso es una precisión binaria.

El objetivo inicial de este trabajo era implementar un sistema de estimación de la pose de una cámara con robustez a objetos dinámicos. Se ha hecho un análisis en detalle de los problemas que pueden surgir en los algoritmos de estimación de pose

de una cámara monocular cuando la escena contiene objetos dinámicos. Se ha visto que el uso de máscaras binarias que detectan objetos dinámicos mejoran mucho los resultados. Por eso se decidió a continuación crear un modelo de red capaz de detectar con únicamente dos frames los objetos dinámicos presentes en la escena. Nos habría gustado probar cómo funciona un sistema de SLAM en el que se filtran los *key points* con las máscaras generadas por nuestra red convolucional y así estudiar el efecto de los objetos dinámicos. Pero motivos temporales no se ha podido realizar este experimentos para este trabajo de fin de grado.

El resto de problemas se han comentado en los apartados correspondientes.

Capítulo 6

Conclusiones

Antes de comenzar este proyecto, desconocía la gran cantidad de problemas que hay que superar para conseguir estimar la posición de una cámara de manera fiable y apenas había ahondado en los temas que abarca la visión por computador fuera del curso impartido en la universidad. Tras haber investigado sobre el tema propuesto por la directora de este proyecto, Berta Bescós, desestimé la influencia de los elementos dinámicos a la hora de estimar una posición, pero ahora comprendo el enorme lastre que estos implican para sistemas como pueden ser la conducción autónoma o los sistemas de SLAM. A lo largo de mi vida académica, no había desarrollado redes neuronales desde cero y tras implementar la red utilizada en este trabajo considero que tengo una base de conocimiento lo suficientemente sólida como para expandir los avances conseguidos en este frente, implementando por mi cuenta las mejoras comentadas en la Sección 4.9.

Este proyecto comenzó con la idea de comprobar si el algoritmo de RANSAC era lo suficientemente bueno como para detectar objetos dinámicos en una escena, o hasta qué punto podría serlo. Se ha estudiado en el Capítulo 3 que este algoritmo carece de la precisión necesaria para hacerlo correctamente. En cuanto a la red neuronal implementada, considero que se pueden llegar a obtener buenos resultados y que se puede alcanzar un rendimiento óptimo para ejecutar en tiempo real el sistema propuesto aunque se necesitaría dedicar más tiempo del asignado para un trabajo de fin de grado.

A este trabajo se le ha dedicado alrededor de 390 horas, de las cuales, aproximadamente 70 horas se han dedicado a documentación sobre el problema e investigación, 260 horas se han dedicado a implementación de código y un tercio de estas a depuración del mismo. Para la redacción de la memoria se han empleado en torno a 60 horas. Al tiempo dedicado a este trabajo habría que sumarle el tiempo de ejecución de los programas implementados, esto añadiría aproximadamente 150 horas extra.

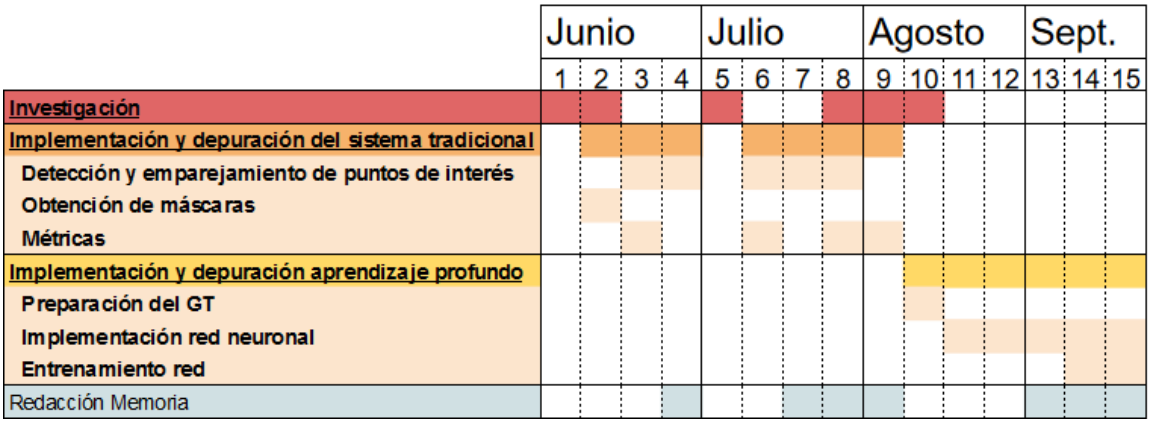


Figura 6.1: Cronograma.

Bibliografía

[1] Wikipedia. Visión artificial — Wikipedia, the free encyclopedia, 2020.

[2] Noah Snavely, Steven M Seitz, and Richard Szeliski. Photo tourism: exploring photo collections in 3d. In *ACM Siggraph 2006 Papers*, pages 835–846. 2006.

[3] Berta Bescos, José M Fácil, Javier Civera, and José Neira. Dynaslam: Tracking, mapping, and inpainting in dynamic scenes. *IEEE Robotics and Automation Letters*, 3(4):4076–4083, 2018.

[4] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

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

[6] David G Lowe. Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. Ieee, 1999.

[7] Christopher G Harris, Mike Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

[8] Jianbo Shi et al. Good features to track. In *1994 Proceedings of IEEE conference on computer vision and pattern recognition*, pages 593–600. IEEE, 1994.

[9] Miroslav Trajković and Mark Hedley. Fast corner detection. *Image and vision computing*, 16(2):75–87, 1998.

[10] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Computer vision and image understanding*, 110(3):346–359, 2008.

[11] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. In *European conference on computer vision*, pages 778–792. Springer, 2010.

[12] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International conference on computer vision*, pages 2564–2571. Ieee, 2011.

[13] Wikipedia. Serie de taylor — Wikipedia, the free encyclopedia, 2020.

[14] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. 1981.

[15] Open CV. Optical flow — OpenCV, Open Source Computer Vision, 2020.

[16] Wikipedia. Pinhole camera model — Wikipedia, the free encyclopedia, 2020.

[17] Wikipedia. Fundamental matrix — Wikipedia, the free encyclopedia, 2020.

[18] Wikipedia. Essential matrix — Wikipedia, the free encyclopedia, 2020.

[19] Wikipedia. Random sample consensus — Wikipedia, the free encyclopedia, 2020.

[20] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on robotics*, 32(6):1309–1332, 2016.

[21] Raul Mur-Artal and Juan D Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[22] Francois Chollet. *Deep Learning with Python.* Manning Publications Co., 2018.

[23] Kaiming He, Georgia Gkioxari, Piotr Dollar, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[24] Daniel Bolya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. Yolact: Real-time instance segmentation. In *ICCV*, 2019.

[25] Wikipedia. Dataflow programming — Wikipedia, the free encyclopedia, 2020.

[26] Wikipedia. Differentiable programming — Wikipedia, the free encyclopedia, 2020.

[27] Kevin Michael Judd and Jonathan D Gammell. The oxford multimotion dataset: Multiple se (3) motions with ground truth. *IEEE Robotics and Automation Letters*, 4(2):800–807, 2019.

[28] Robert Laganière. *OpenCV 3 Computer Vision Application Programming Cookbook*. Packt Publishing Ltd, 2017.

[29] Wikipedia. Eight-point algorithm — Wikipedia, the free encyclopedia, 2020.

[30] Xin-Yu Kuo, Chien Liu, Kai-Chen Lin, and Chun-Yi Lee. Dynamic attention-based visual odometry. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 36–37, 2020.

[31] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.

[32] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick van der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

Lista de Figuras

Anexo A

Código

A.0.1. Pose relativa

Código utilizado para estimar la pose explicado en la Subsección 3.4.1.

```
// Funci n encargada de recuperar la pose a partir de la matriz fundamental.
cv::Mat computePose(
    cv::Mat fundamental, cv::Mat K, std::vector<cv::Point2f> points_im1,
    std::vector<cv::Point2f> points_im2)
  {
  cv::Mat essential =K.t() * fundamental * K;

  cv::Mat R, t;
  cv::recoverPose(essential, points_im1, points_im2, K, R, t);

  cv::Mat pose(4, 4, CV_64F, cv::Scalar(0));
  // Primera columna
  pose.at<double>(0, 0) = R.at<double>(0,0);
  pose.at<double>(1, 0) = R.at<double>(1,0);
  pose.at<double>(2, 0) = R.at<double>(2,0);
  pose.at<double>(3, 0) = 0;
  // Segunda columna
  pose.at<double>(0, 1) = R.at<double>(0,1);
  pose.at<double>(1, 1) = R.at<double>(1,1);
  pose.at<double>(2, 1) = R.at<double>(2,1);
  pose.at<double>(3, 1) = 0;
  // Tercera columna
  pose.at<double>(0, 2) = R.at<double>(0,2);
  pose.at<double>(1, 2) = R.at<double>(1,2);
  pose.at<double>(2, 2) = R.at<double>(2,2);
  pose.at<double>(3, 2) = 0;
  // Cuarta columna
  pose.at<double>(0, 3) = t.at<double>(0);
  pose.at<double>(1, 3) = t.at<double>(1);
  pose.at<double>(2, 3) = t.at<double>(2);
  pose.at<double>(3, 3) = 1;

  return  pose;
}



// Pose en el primer frame
cv::Mat T01 = getPoseFromGT(TFG::to_string(i));
// Pose en el segundo frame
cv::Mat T02 = getPoseFromGT(TFG::to_string(i+step));
// Pose relativa entre ambos frames
cv::Mat pose = T01.inv() * T02;
// Pose estimada mediante RANSAC
cv::Mat estimatedPose =TFG::computePose(fundamental, K, ptsIm1, ptsIm2);

cv::Mat error = estimatedPose * pose;
cv::Point3f t(m.to3D(error.col(3)));
double translationError = cv::sqrt(pow(t.x,2) + pow(t.y, 2) + pow(t.z, 2));
```

A.0.2. Máscaras

Código utilizado para extraer las máscaras explicado en la Sección 3.3.

```
void TFG::mask::firstStep(std::string frameNumber, float scale_factor) {
    // Obtener la posici n de la camara respecto del mundo
    cv::Mat twc = getTwc("sensor_payload", frameNumber);
    for( int i = 0; i < points.size(); i++){
    // Cargar los puntos asignados a cada objeto
        std::vector<cv::Point3f> puntos = points[i];
        std::string item = items[i];
    // Obtener la posici n del objeto respecto del mundo
        cv::Mat two = getTwc(item, frameNumber);
        std::vector<cv::Mat> Xo_;
        for( cv::Point3f  punto : puntos){
            float x = punto.x;
            float y = punto.y;
            float d = punto.z;
            cv::Mat Xc = getXc(x, y, d);
            cv::Mat Xw = twc * Xc;
            Xo_.push_back(two.inv() * Xw);

        }
        Xo.push_back(Xo_);
    }
}

void TFG::mask::secondStep(std::string frame ) {

    // Obtener la posici n de la camara respecto del mundo
    cv::Mat twc = getTwc("sensor_payload", frame);

    std::vector<std::vector<cv::Point2f>> aux;


    for( int i = 0; i < points.size(); i++){
        std::vector<cv::Mat> Xo_ = Xo[i];
        std::string item = items[i];
    // Obtener la posici n del objeto respecto del mundo
        cv::Mat two = getTwc(item, frame);
        std::vector<cv::Point2f> output_;
        for(auto pto : Xo_){
            cv::Mat Xw = two * pto;
            cv::Mat Xc = to3D(twc.inv() * Xw);

            float d = (float) Xc.at<double>(2, 0);
            Xc = Xc / d;

            cv::Mat X = K * Xc;
            float u = static_cast<float>(X.at<double>(0, 0));
            float v = static_cast<float>(X.at<double>(1, 0));
            cv::Point2f p(u, v);
            output_.push_back(p);
        }
        aux.push_back(output_);
    }
    output = aux;

}

cv::Mat TFG::mask::getXc(float u, float v, float d) {
    cv::Mat uv1 = cv::Mat::zeros(3, 1, CV_64F);

    uv1.at<double>(0, 0) = u;
    uv1.at<double>(1, 0) = v;
    uv1.at<double>(2, 0) = 1;

    cv::Mat xy1 = (K.inv() * uv1);

    return toHomogeneous(xy1 * d);
}


```

```
cv::Mat TFG::mask::toHomogeneous(cv::Mat input){
    cv::Mat output = cv::Mat::zeros(4,1,input.type());
    output.at<double>(0,0) = input.at<double>(0,0);
    output.at<double>(1,0) = input.at<double>(1,0);
    output.at<double>(2,0) = input.at<double>(2,0);
    output.at<double>(3,0) = 1;
    return output;
}

cv::Mat TFG::mask::to3D(cv::Mat input){
    cv::Mat output = cv::Mat::zeros(3,1,input.type());
    cv::Mat aux = normalize(input);
    output.at<double>(0,0) = aux.at<double>(0,0);
    output.at<double>(1,0) = aux.at<double>(1,0);
    output.at<double>(2,0) = aux.at<double>(2,0);
    return output;
}

cv::Mat TFG::mask::normalize(cv::Mat input){
    cv::Mat output;
    float factor = static_cast<float>(input.at<double>(3, 0));
    output = input;
    if (factor != 0.0f){
        output /= factor;
    }
    return output;
}

cv::Mat TFG::mask::genMask(const cv::Mat &img){


    cv::Mat mask = img.clone();
    for( auto contorno_00 : output){
        for(auto cara : caras){
            std::vector<cv::Point> contorno;
            for(int i = 0 ; i < cara.size() - 1 ; i++){
                contorno.push_back(
          cv::Point(contorno_00[cara[i]].x,
          contorno_00[cara[i]].y)
        );
            }
            const cv::Point* inicial = &contorno.at(0);
            const cv::Point** puntos = {&inicial};
            int numberOfPoints = (int)contorno.size();

            cv::fillPoly (
        mask,
        puntos,
        &numberOfPoints,
        1,
        cv::Scalar (255, 255, 255),
        8
      );

        }
    }
    return mask;
}
```

A.0.3. Emparejamiento robusto

Código utilizado para implementar el emparejamiento robusto en Sección 3.2.

```
cv::Mat robustMatcher::ransacTest(
    const std::vector<cv::DMatch>& matches, std::vector<cv::KeyPoint>& kp1,
    std::vector<cv::KeyPoint>& kp2, std::vector<cv::DMatch>& outMatches,
    std::vector<cv::Point2f>& p1, std::vector<cv::Point2f>& p2 ) )
  {
  outMatches.clear();
  outliers1.clear();
  outliers2.clear();
  inliers1.clear();
  inliers2.clear();
  p1.clear();
  p2.clear();


  // 1. Convertir KeyPoints en Point2f
  std::vector<cv::KeyPoint> p1_, p2_;
  for(auto it = matches.begin(); it != matches.end(); ++it){
    // KP de la izda
    p1.push_back(kp1[it->queryIdx].pt);
    p1_.push_back(kp1[it->queryIdx]);
    // KP de la dcha
    p2.push_back(kp2[it->trainIdx].pt);
    p2_.push_back(kp2[it->trainIdx]);
  }

  // 2. Calcular la matriz fundamental usando RANSAC
  std::vector<uchar> mask(p1.size(), 0);
  cv::Mat fundamental =
    cv::findFundamentalMat(p1, p2, mask, cv::FM_RANSAC, distance, confidence);


  // 3. Obtener los inliers (matches restantes)
  auto it_mask = mask.begin();
  auto it_matches = matches.begin();
  for( ; it_mask != mask.end(); ++it_mask, ++it_matches){
    // Es inlier
    if(*it_mask){
      outMatches.push_back(*it_matches);
    }
  }



  for(int i = 0; i < mask.size(); i++){
    if(!mask[i]) {
      outliers1.push_back(p1_[i]);
      outliers2.push_back(p2_[i]);
    } else {
      inliers1.push_back(p1_[i]);
      inliers2.push_back(p2_[i]);
    }
  }


  // 4. Usa los matches calculados para re-estimar la matriz fundamental
  //    usando el algoritmo de 8 puntos
  if(refineF){
    p1.clear();
    p2.clear();


    // 4.1 Convertir KeyPoints en Point2f
    for(auto it = outMatches.begin(); it != outMatches.end(); ++it){
      // KP de la izda
      p1.push_back(kp1[it->queryIdx].pt);
      // KP de la dcha
      p2.push_back(kp2[it->trainIdx].pt);
    }

```

```
    // 4.2 Calcular la matriz fundamental mediante el algoritmo de 8 puntos
    auto _8pts= cv::findFundamentalMat(p1, p2, cv::FM_8POINT);
    if(_8pts.rows * _8pts.cols == 9){
      fundamental = _8pts;
    }
  }

  // 5. Usar la matriz fundamental calculada para re-estimar los emparejamientos
  if(refineM){
    p1.clear();
    p2.clear();
    // 4.1 Convertir KeyPoints en Point2f
    for(auto it = outMatches.begin(); it != outMatches.end(); ++it){
      // KP de la izda
      p1.push_back(kp1[it->queryIdx].pt);
      // KP de la dcha
      p2.push_back(kp2[it->trainIdx].pt);
    }
    std::vector<cv::Point2f> np1, np2;

    // Corregir las coordenadas de los puntos de inter s calculados
    cv::correctMatches(fundamental, p1, p2, np1, np2);
    int i = 0;
    for(auto it = outMatches.begin(); it != outMatches.end(); ++it){
      // KP de la izda
      kp1[it->queryIdx].pt = np1[i];
      // KP de la dcha
      kp2[it->trainIdx].pt = np2[i];
      i++;
    }
  }
  return fundamental;
}

cv::Mat robustMatcher::match(
    const cv::Mat& im1, const cv::Mat& im2, std::vector<cv::DMatch>& matches,
    std::vector<cv::KeyPoint>& kp1, std::vector<cv::KeyPoint>& kp2,
    std::vector<cv::Point2f>& p1, std::vector<cv::Point2f>& p2)
  {
  // 1. Detectar KP
  detector->detect(im1, kp1);
  detector->detect(im2, kp2);
  cv::Mat test1;

  // 2. Computar descriptores
  cv::Mat d1, d2;
  descriptor->compute(im1,kp1, d1);
  descriptor->compute(im2,kp2, d2);


  // 3. Calcular coincidencias
  std::vector<cv::DMatch> outputMatches;
  cv::BFMatcher matcher(normType /* Distancia */, false /* CrossCheck */);
  std::vector<std::vector<cv::DMatch>> vecino;
  matcher.knnMatch(d1, d2, vecino, 2);
  for (int i = 0; i < vecino.size(); i++) {
    if (vecino[i][0].distance < ratio * vecino[i][1].distance) {
      outputMatches.push_back(vecino[i][0]);
    }
  }

  // 4. Validar matches usando RANSAC
  cv::Mat fundamental = ransacTest(outputMatches, kp1, kp2, matches,p1, p2);

  return fundamental;
}
```

A.0.4. Red neuronal

El código utilizado para implementar la red neuronal descrita en Capítulo 4 se encuentra en el siguiente repositorio de github. En este repositorio se encuentran las instrucciónes para su correcta ejecución.

`https://github.com/sisques/TFG`