Universidad Zaragoza

Trabajo Fin de Grado

Redes neuronales profundas para mejorar la robustez de ORB-SLAM2

Deep neural networks to improve the robustness of ORB-SLAM2

Autor

Sergio Izquierdo Barranco

Director

Juan Domingo Tardós Solano

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2019

AGRADECIMIENTOS

Me gustaría agradecer la dedicación recibida por parte de Juan D. Tardós, que a lo largo de este trabajo siempre ha estado muy implicado y dispuesto a ayudarme a resolver todos los problemas que se me presentaron.

El desarrollo de este proyecto se ha podido llevar a cabo gracias a la donación de una GPU Titan V por parte de NVIDIA y a la financiación recibida por la Cátedra Mobility City.

RESUMEN

Los sistemas de SLAM (*Simultaneous Localization And Mapping*) permiten la generación de un mapa sobre un entorno desconocido mientras se calcula la localización del agente en ese mismo espacio. El sistema ORB-SLAM2, desarrollado por la Universidad de Zaragoza, permite la generación de estos mapas utilizando cámaras como únicos sensores. Para ello, el sistema se basa en una extracción exhaustiva de puntos de interés ORB que posteriormente, y mediante su emparejamiento entre distintas imágenes, le servirán para conocer el desplazamiento del agente. Sin embargo, la utilización del extractor ORB para este propósito implica que no se utiliza ningún conocimiento empírico del entorno en el que se mueve el agente, resultando en que no todos los puntos extraídos son idóneos para el sistema.

Este trabajo aprovecha los grandes avances en el aprendizaje profundo para superar las limitaciones del sistema original añadiendo un componente de inteligencia mediante el aprendizaje con redes neuronales. Con este objetivo hemos intentado, en primer lugar, añadir este aprendizaje como una capa por encima del extractor ORB para predecir la robustez de los puntos de interés a partir de la localización del punto en la imagen, su descriptor y los niveles de gris de la región donde aparece el punto. El resultado obtenido es negativo, lo que nos permite concluir que la falta de robustez de los puntos ORB depende de factores aleatorios que no es posible aprender.

Con esta conclusión se ha decidido en segundo lugar sustituir completamente tanto la extracción como el emparejamiento de los puntos en el sistema, implementando de nuevo uno de los principales módulos de ORB-SLAM2. En este nuevo sistema creado, una red neuronal marca las zonas de interés de una imagen, y un nuevo sistema de seguimiento se encarga de rastrearlas entre distintos fotogramas. Estas mejoras consiguen que el porcentaje de puntos rastreados correctamente por el sistema cambie drásticamente del 3 % al 56 %.

Índice

1. Introducción y objetivos **1**

1.1. Motivación 2

1.2. Objetivos 3

2. El Sistema ORB-SLAM2 **4**

2.1. Estructura del Programa 4

2.2. ORB Features 5

2.2.1. Extracción 5

2.2.2. Emparejamiento 6

2.3. Análisis del funcionamiento actual 7

3. Aprendizaje de la Robustez **8**

3.1. Creación y Análisis del Dataset 8

3.1.1. Creación 8

3.1.2. Análisis 9

3.2. Diseño de un Perceptrón Multicapa 11

3.2.1. Resultados 12

3.3. Diseño de una Red Convolucional 13

3.3.1. Resultados 14

3.4. Discusión 15

4. Aprendizaje de las Features **16**

4.1. La red SfMLearner 16

4.2. Extracción de Features 18

4.2.1. Análisis de las capas intermedias 18

4.2.2. Elección de Features 19

4.2.3. Integración de TensorFlow en C++ 21

4.3. Seguimiento de las Features 21

4.3.1. Seguimiento por correlación 21

4.3.2. Seguimiento por Lucas-Kanade 22

4.3.3. Cómo sobrevivir a las curvas 24
4.4. Resultados 25

5. Conclusiones **27**
5.1. Trabajo Futuro 27

Anexos **30**

A. Herramientas Utilizadas **31**

B. Gestión del Proyecto **32**

Capítulo 1

Introducción y objetivos

Los sistemas de SLAM (*Simultaneous Localization And Mapping*) permiten la generación de un mapa sobre un entorno desconocido mientras se calcula la localización del agente en ese mismo espacio. De ese problema original se han popularizado debido a su bajo coste las soluciones de visual SLAM, que abordan el problema utilizando cámaras como único sensor.

En este tipo de sistemas existen dos componentes diferenciadas y cuya implementación puede variar de una solución a otra. En primer lugar existe un *feature extractor*, que a partir de una imagen, extrae los puntos o las zonas más útiles para el funcionamiento del sistema. La segunda componente es la encargada de recibir como entrada esas *features* y utilizarlas para realizar el SLAM en sí mismo.

En ORB-SLAM2 [1], el sistema desarrollado por la Universidad de Zaragoza, se hace uso de puntos ORB para la extracción de *features* mientras que el SLAM se realiza mediante el emparejamiento de esos puntos y siguiendo los conocidos principios de la geometría multivista [2]. El hecho de utilizar este extractor hace que no se aproveche ningún conocimiento sobre el entorno en el que se mueve el agente, algo que otras aproximaciones, como SfMLearner [3], han intentado solucionar haciendo uso del aprendizaje automático y redes neuronales profundas, Fig. 1.1. Sin embargo, en esta segunda aproximación se obliga a la red a aprender conceptos complejos cuya solución podría ser obtenida utilizando principios geométricos.

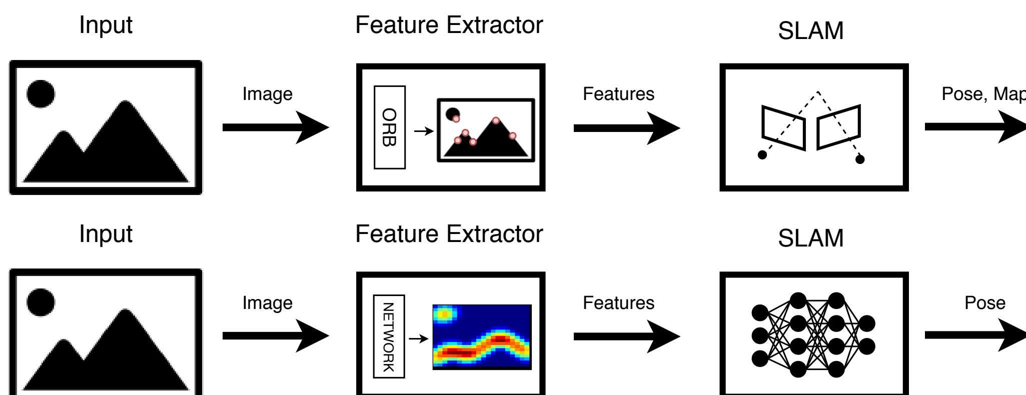


Figura 1.1: En ORB-SLAM2 (arriba) la extracción de *features* se realiza con ORB y el SLAM con geometría multivista. En SfMLearner (abajo), una red neuronal se encarga de realizar ambas tareas.

El objetivo de este trabajo es aprovechar las fortalezas de estas dos aproximaciones para mejorar ORB-SLAM2, manteniendo aquellos cálculos que aprovechan los estudiados principios geométricos de la visión por computador, pero añadiendo un componente de inteligencia y conocimiento del entorno a la extracción de puntos de interés mediante el uso de redes neuronales profundas.

1.1. Motivación

El éxito del funcionamiento de un sistema de SLAM basado en la *multiview geometry* recae tanto en la calidad de los puntos extraídos como en la robustez de sus emparejamientos. Esto causa que por ejemplo, los sistemas tengan especiales dificultades para afrontar escenas con gran dinamismo, en las que las *features* extraídas no son rígidas y perjudican los algoritmos de optimización. También son sensibles a estructuras con superficies difusas, ya que al carecer de textura, el emparejamiento de los puntos no siempre se efectúa correctamente, afectando a la precisión del sistema.

En ORB-SLAM2 se utiliza un extractor de *features* que trata de minimizar el efecto de las superficies difusas marcando como puntos de interés solo aquellos que tienen mucha textura, por ejemplo esquinas. Este método es meramente ingenieril y aunque ofrece resultados satisfactorios en lo relativo a la precisión de los emparejamientos, la calidad de las extracciones varía, resultando en que los puntos escogidos no siempre son idóneos para ser rastreados. De hecho, nuestros experimentos demuestran que solamente el 3 % de los puntos extraídos acaban siendo útiles para el sistema. Hay muchas razones que pueden hacer que un punto no pueda ser seguido correctamente, (objetos móviles, oclusiones, cambios de iluminación, desaparición del campo visual...), pero anticipar estas causas es una tarea compleja y es difícil que pueda ser tratada sin emplear la experiencia y el aprendizaje.

Incluso en escenas complejas con todas esas características, el ser humano demuestra una increíble capacidad para deducir la estructura tridimensional y su localización espacial a partir de sus receptores visuales. Aunque esta capacidad se debe en gran medida a la estereopsis (visión binocular) y al paralaje en movimiento, también es posible extraer esa información de imágenes estáticas, empleando conocimientos previos adquiridos con la experiencia, como el tamaño de objetos familiares o su perspectiva [4]. De la misma forma en la que el ser humano se ayuda de conocimientos empíricos en su percepción, los sistemas de SLAM podrían aprovecharse aprendiendo acerca de los entornos a los que se van a enfrentar con la ayuda del aprendizaje profundo .

No obstante, esto no quiere decir que la solución más conveniente sea prescindir del sistema actual y emplear aprendizaje automático de forma *end-to-end* sustituyendo todo por redes neuronales. La solución óptima es, posiblemente, una simbiosis entre las dos técnicas de forma que se complementen. Un sistema en el que el aprendizaje automático sirva para reforzar la robustez en aquellos eslabones estratégicos en los que la geometría encuentra dificultades y problemas ante la aparición de escenas complejas.

1.2. Objetivos

A lo largo de este proyecto se van a estudiar e implementar dos posibles estrategias para añadir aprendizaje profundo al sistema ORB-SLAM2 actual. Las dos aproximaciones tratan de sustituir, modificar o complementar el módulo de extracción de *features*, manteniendo los algoritmos de SLAM existentes. En primer lugar, se intentará añadir la componente de inteligencia como una capa por encima del extractor ORB, diseñando un sistema que sea capaz de predecir la utilidad y robustez de los puntos seleccionados, capítulo 3. En la segunda aproximación, se prescinde del extractor ORB actual, sustituyendo completamente este módulo por una red neuronal profunda que marque las zonas de interés de una imagen, capítulo 4. Como resultado de una modificación tan importante, también será necesario reimplementar el sistema de emparejamientos. La figura 1.2 muestra un esquema de estas dos nuevas aproximaciones en contraste con las presentadas en la figura 1.1.

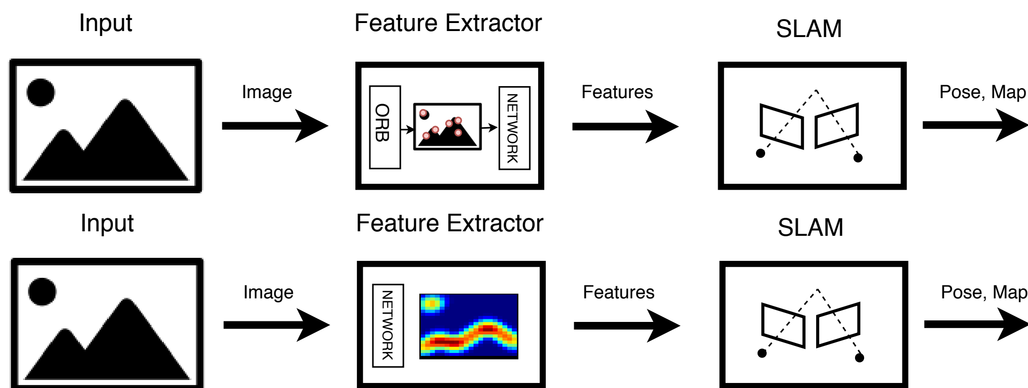


Figura 1.2: En la primera aproximación (arriba) se intentó utilizar una red para decidir qué puntos son provechosos, en la segunda (abajo) la red responde directamente con las zonas de interés. El SLAM en sí mismo se mantiene utilizando la geometría multivista.

De forma más concreta, en este trabajo se tienen los siguientes objetivos:

1. Estudio de técnicas de SLAM y el estado del arte en redes profundas.
2. Generación de un dataset a partir de datos de robustez de *features* de ORB-SLAM2.
3. Diseño y entrenamiento de una red neuronal para predecir la robustez de los puntos ORB.
4. Evaluación de la calidad y viabilidad de estas predicciones
5. Diseño y entrenamiento de una red neuronal para detectar *features*.
6. Integración de la red para extraer *features* en ORB-SLAM2.
7. Evaluación de prestaciones y comparación con ORB-SLAM2 original.

Capítulo 2

El Sistema ORB-SLAM2

El sistema ORB-SLAM2 se basa en la observación de puntos de interés o *features* desde distintas posiciones de cámara para que mediante su emparejamiento sea posible estimar tanto la posición de los puntos observados como la de las cámaras. Una de sus claves es que evita la inclusión de información redundante eligiendo qué fotogramas son más relevantes, $Keyframes$, y qué puntos son interesantes para el mapa creado, $MapPoints$, lo que le permite ejecutar optimizaciones con *Bundle Adjustement* [5] en tiempo real.

2.1. Estructura del Programa

En el sistema, tres hilos de ejecución trabajan de forma coordinada para ejecutar el ciclo de creación y procesamiento de los puntos de forma continua (Fig. 2.1):

- Tracking: Se encarga de procesar cada nueva imagen, extrayendo sus *features* y realizando emparejamientos con las del fotograma anterior y con los *MapPoints* a su alrededor para estimar la posición de la cámara. Si esta imagen contiene información novedosa se convertirá en un *Keyframe* y se insertará al mapa.
- Local Mapping: A partir de los *Keyframes* insertados se intentan triangular nuevos puntos emparejando las *features* de estos fotogramas que todavía no están asociadas a ningún *MapPoint*. Las triangulaciones realizadas correctamente se insertarán como nuevos puntos del mapa y se realizará una optimización del grafo local, *Local Bundle Adjustement*.
- Loop Closing: Su misión principal es determinar cuándo la cámara está pasando por un lugar ya visitado para poder cerrar un ciclo en el grafo interno y realizar una optimización global, *Global Bundle Adjustement*.

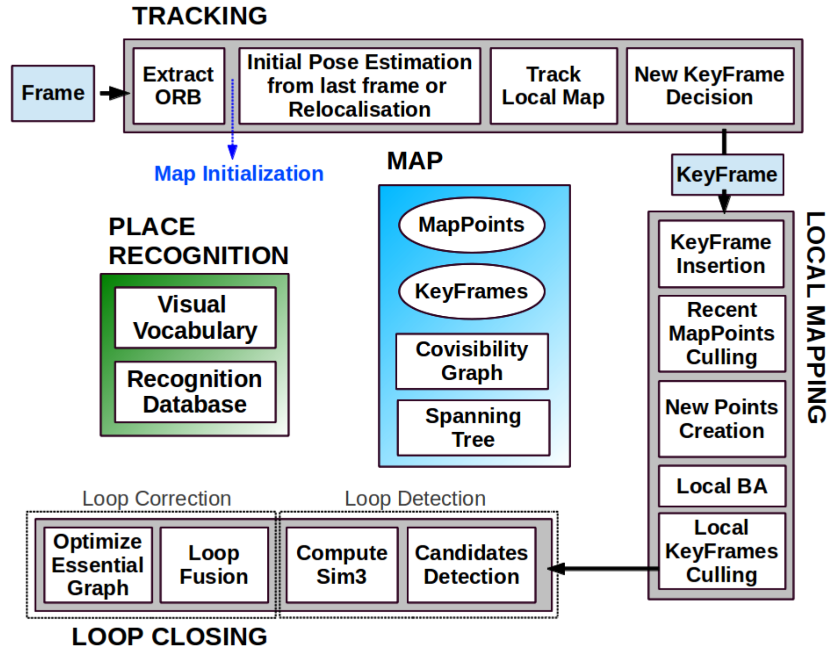


Figura 2.1: Estructura del sistema, mostrando las tareas realizadas por cada uno de los hilos. Extraída de [6].

2.2. ORB Features

Dado que ORB-SLAM2 es un sistema basado en el emparejamiento de puntos de interés, el módulo de extracción de *features* no solo debe seleccionar y devolver puntos de calidad, si no que además tiene que proveer un algoritmo que permita su emparejamiento de forma rápida y fiable. El extractor ORB (Oriented FAST and rotated BRIEF) [7], el usado en ORB-SLAM2, ofrece estas características basando su extracción en los puntos FAST [8], el emparejamiento en los descriptores BRIEF [9] y añadiendo mejoras que permiten que funcione de forma más robusta.

2.2.1. Extracción

Para que se puedan realizar emparejamientos entre distintas imágenes el extractor debe responder ante los mismos puntos aunque no tengan exactamente la misma apariencia. Esta propiedad, conocida como invarianza, puede referir a distintas situaciones (cambios de iluminación, de orientación, de tamaño o de perspectiva) y en FAST se intenta conseguir buscando puntos con gran distintividad: píxeles claros sobre fondos oscuros o viceversa, Fig 2.2. Esta estrategia es mejorada en ORB realizando la extracción a diferentes escalas, lo que mejorar la invarianza a cambios de tamaño, y con una ordenación final de los puntos en función de la respuesta de Harris [10] para quedarse solo con los mejores y eliminar aristas. El estar basado en sencillas comparaciones de brillo entre píxeles hace que su ejecución sea muy rápida en comparación con otros descriptores más sofisticados y que presente buena invarianza ante cambios en la iluminación.

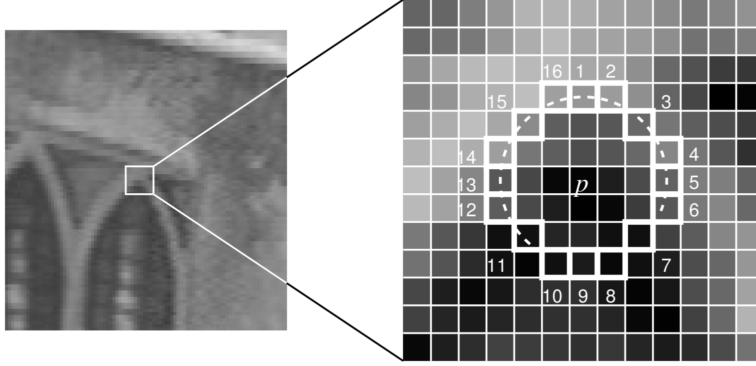


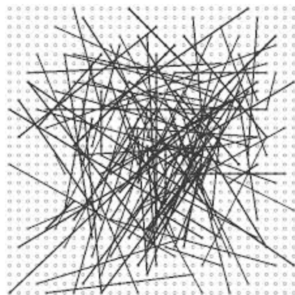
Figura 2.2: En FAST, para que p sea considerado un punto de interés, al menos n vecinos seguidos deben ser más brillantes o más oscuros que p con un umbral. En la imagen los píxeles con un cuadrado blanco marcan los vecinos a considerar para este test. La línea discontinua en blanco marca puntos vecinos seguidos más claros que p con un umbral. Extraído de [8].

2.2.2. Emparejamiento

El emparejamiento de los puntos de interés extraídos se realiza mediante el uso de descriptores, unos vectores que identifican las características del punto y sus alrededores, de forma que si dos puntos tienen descriptores similares es probable que se trate del mismo. En BRIEF se calcula un descriptor binario basado en la comparación de intensidades de 256 pares de píxeles alrededor del punto, obteniendo un descriptor de tamaño 8 bytes, Fig 2.3, que puede ser comparado utilizando la distancia de Hamming.

En ORB se añade a estos descriptores invarianza a la rotación teniendo en cuenta la orientación del punto, obtenida como el ángulo del vector desde el centro de la esquina hasta el centroide del parche, para las comparaciones entre píxeles. Para ello, las coordenadas de los píxeles a comparar se trasladan de acuerdo al ángulo calculado, lo que tiene el mismo efecto que rotar el parche de la imagen.

La principal característica de este descriptor es su velocidad tanto al computarse como al emparejar, ya que la distancia de Hamming binaria se puede implementar con un simple XOR, una rápida operación de bajo nivel.



$$\tau(\mathbf{p};\mathbf{x},\mathbf{y}) := \begin{cases} 1 & \text{if } \mathbf{p}(\mathbf{x}) < \mathbf{p}(\mathbf{y}) \\ 0 & \text{otherwise} \end{cases}$$

$$f_{n_d}(\mathbf{p}) := \sum_{1 \leq i \leq n_d} 2^{i-1} \tau(\mathbf{p};\mathbf{x},\mathbf{y})$$

Figura 2.3: En BRIEF se comparan las intensidades de los n_d píxeles que rodean a un punto de interés, en la práctica $n_d = 256$. La imagen de la izquierda muestra los píxeles que son comparados. El descriptor f_{n_d} se obtiene concatenando cada una de esas comparaciones. Extraído de [9].

2.3. Análisis del funcionamiento actual

Hemos llevado a cabo un análisis del módulo de extracción y emparejamiento de los puntos de interés en el sistema actual con el objetivo de crear una referencia para el resto del trabajo y poder comparar los resultados obtenidos. Este análisis se ha realizado ejecutando las 11 primeras secuencias del KITTI dataset [11] utilizando una configuración monocular.

Durante las ejecuciones, se han recolectado las estadísticas de todos los puntos generados distinguiendo entre 4 situaciones distintas que hacen que un punto sea provechoso o no. Las 4 situaciones consideradas, junto a su porcentaje de ocurrencias son:

1. No emparejado: 94.84 %
2. Emparejado, pero marcado como espurio por un optimizador: 1.51 %
3. Emparejado y correcto, pero considerado redundante: 0.68 %
4. Emparejado, correcto y provechoso: 2.97 %

A efectos prácticos se puede diferenciar entre puntos inútiles (las tres primeras situaciones) y útiles (la última). La causa de la gran mayoría de los puntos inútiles es que no han podido ser emparejados, lo que puede ser debido a una exagerada extracción de puntos en sitios poco distintivos. En la figura 2.4 se observa la abrumadora mayoría de puntos inútiles en una imagen cualquiera, muchos de ellos extraídos en localizaciones desacertadas como el asfalto del suelo o la vegetación homogénea de la izquierda.

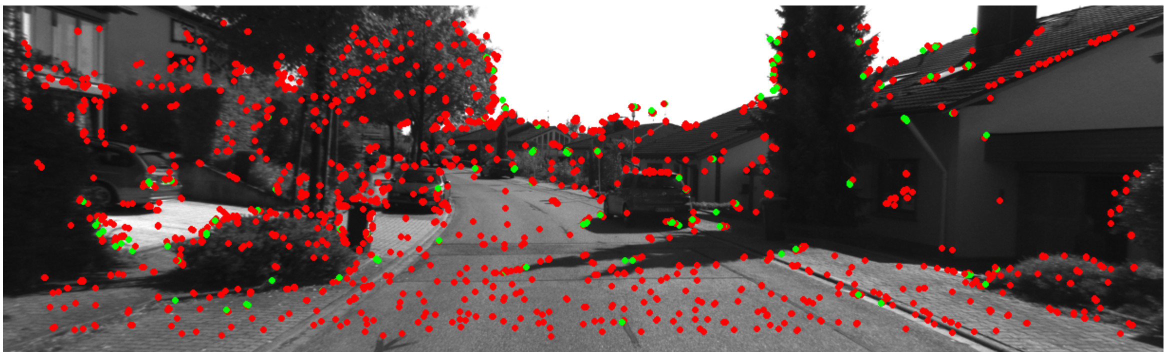


Figura 2.4: En el sistema actual la mayoría de los puntos extraídos no acaban siendo útiles (rojo) y solo unos pocos consiguen ser rastreados (verde).

Capítulo 3

Aprendizaje de la Robustez

En la primera aproximación planteada, el conocimiento y la inteligencia del entorno se han intentado añadir al sistema como un complemento al extractor de *features* (figura 1.1). De forma más concreta, nuestro objetivo ha sido diseñar una red neuronal profunda que tome como entrada cada punto extraído por ORB y prediga la utilidad que tendrá para el sistema de SLAM en función de su robustez, de forma que se evite la inserción de puntos espurios y se empleen más recursos en emparejar los puntos que han sido valorados como aprovechables.

3.1. Creación y Análisis del Dataset

Para realizar el aprendizaje se necesita un conjunto de datos de entrenamiento, por lo que el primer paso ha sido generar un dataset recolectando características de las *features* de ORB-SLAM2.

3.1.1. Creación

De toda la información disponible en los puntos de interés, hemos considerado que las características más relevante que pueden ser utilizadas para aprender la utilidad de estos, y que por ello deben ser incluidas en el dataset, son:

- Coordenadas en la imagen
- Escala de extracción
- Orientación
- Descriptor binario (256 bits)
- Ventana de la imagen alrededor del punto de tamaño 32×32 píxeles.

Para su generación, hemos creado y ejecutado una versión modificada de ORB-SLAM2 que guarda en ficheros todas estas características de cada uno de los puntos extraídos así como su etiqueta: bueno (punto emparejado, correcto y provechoso) o malo (resto). La ejecución se ha realizado sobre las mismas 11 secuencias empleadas en el análisis del funcionamiento de ORB-SLAM2 (apartado 2.3) para que se puedan realizar comparaciones.

Estos datos fueron mezclados de forma aleatoria y separados en tres conjuntos distintos: entrenamiento (80 %), validación (10 %) y test (10 %). Además, para que la red pueda aceptar la diversa naturaleza de los distintos datos de cada una de las dimensiones se estandarizó cada uno de los conjuntos (conforme a sus propios datos) para que tuviesen media 0 y desviación típica 1.

3.1.2. Análisis

Al estudiar el recién creado dataset se ha observado, como era de esperar, que está tremendamente desequilibrado: de los 21.677.024 puntos extraídos solo 618.400 han sido etiquetados como buenos, menos del 3 %. Con el objetivo de encontrar las características más discriminantes de las muestras positivas y así poder diseñar un modelo de red adaptado al problema del desequilibrio se hizo un análisis de las distintas dimensiones extraídas.

En la primera observación, analizando la distribución de la orientación (figura 3.1) parece que no hay grandes diferencias entre puntos buenos y malos, ambos siguen unos patrones muy similares, concentrando la mayor parte de las orientaciones alrededor de 90º y 270º. De igual forma, en la figura 3.2, se observa cómo la proporción de ambas clases en cada escala es también muy parecida.

En la figura 3.3 sí que se aprecia una diferencia en la distribución espacial de los puntos. Por ejemplo, el disco que rodea al horizonte parece ser una buena fuente de puntos, especialmente en la zona superior. Estas áreas son, por ejemplo, edificios a una distancia suficiente como para que los puntos se vean en bastantes fotogramas pero no tan lejos como para que no haya paralaje y no se pueda estimar su profundidad. Todo lo contrario pasa en los bordes de la imagen, donde los puntos desaparecen rápidamente, y en el horizonte, donde los puntos no se llegan a observar con suficiente paralaje. En general, parece haber más puntos buenos a la izquierda, algo que puede ser debido a que los vídeos han sido grabados desde un automóvil circulando por el carril derecho.

Aunque parece que las coordenadas pueden ser discriminantes para seleccionar si un punto es bueno o no, es conveniente tener en cuenta que en las zonas con mayor concentración el porcentaje de puntos buenos no llega al 8 %.

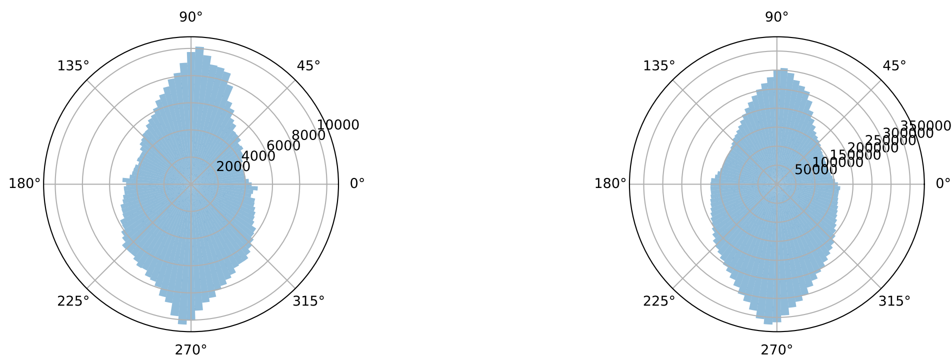


Figura 3.1: Histograma de las orientaciones de los puntos buenos (izquierda) y malos (derecha)

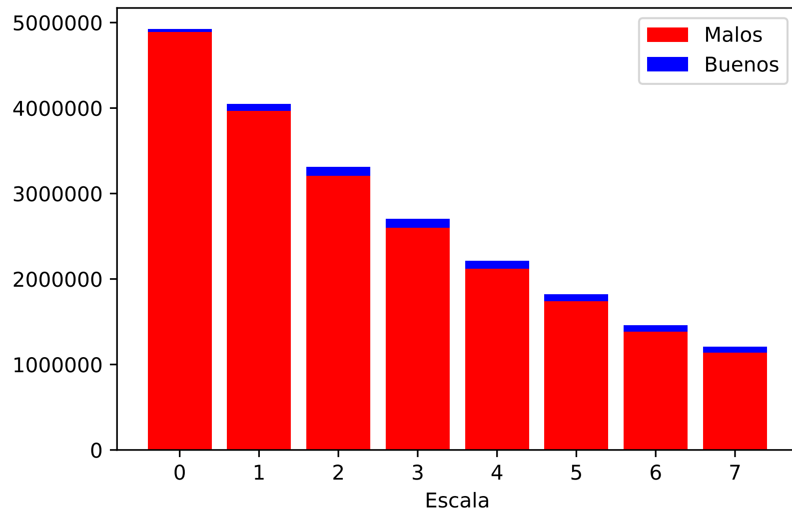


Figura 3.2: Puntos buenos y malos extraídos en cada escala. El hecho de que en cada escala superior haya menos extracciones es intencionado.

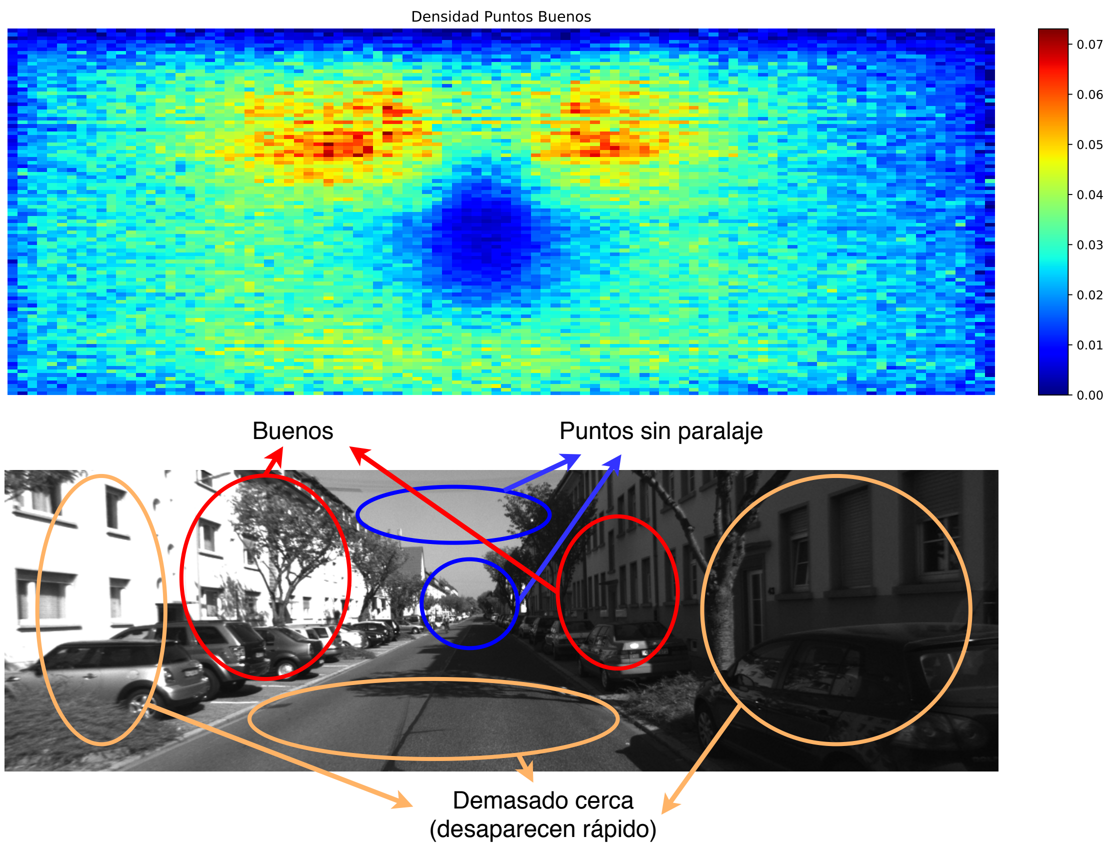


Figura 3.3: Arriba: Porcentaje de puntos buenos en cada zona de las imágenes. Creado a partir de todos los puntos extraídos. Abajo: Posible explicación de la densidad en las distintas zonas para una imagen de muestra de las secuencias.

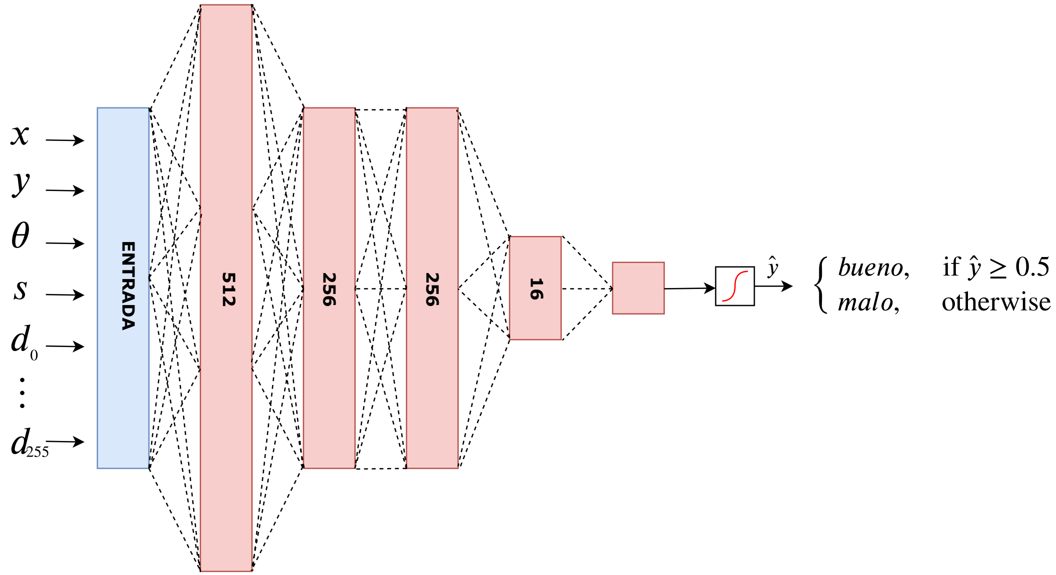


Figura 3.4: Esquema de la arquitectura del mejor modelo de perceptrón multicapa obtenido.

3.2. Diseño de un Perceptrón Multicapa

El primer modelo de red neuronal diseñado y probado fue un clasificador binario utilizando un perceptrón multicapa, el modelo más sencillo de red neuronal, compuesto de múltiples capas densas (totalmente conectadas) de neuronas artificiales. Como los perceptrones no son el modelo más indicado para ser usados con imágenes se evitó aumentar la complejidad del modelo y el número de pesos descartando los parches de los puntos y utilizando únicamente el resto de datos del dataset (coordenadas, orientación, escala y descriptor).

Al utilizar un conjunto de datos muy desequilibrado, fijarse en la tasa de aciertos de la red, la *accuracy*, no refleja el correcto funcionamiento, pues se tendrá una estadística muy buena aunque el sistema no funcione (una red que rechace todos los puntos, tendrá una tasa de aciertos del 97 %). Por ello se utilizan tres medidores extra: la *precision*, que refleja qué fracción puntos de los marcados como buenos efectivamente lo son, el *recall*, qué fracción de los puntos buenos que había han sido encontrados, y el F_1 *score* que es la media armónica de los anteriores (eq 3.1), y penaliza valores bajos de ambos, Eq 3.1.

$$precision = \frac{\text{true positive}}{\text{true positive} + \text{false positive}} \qquad recall = \frac{\text{true positive}}{\text{true positive} + \text{false negative}}$$

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{3.1}$$

Además es probable que las muestras positivas no consigan tener influencia sobre la función de coste por ser minoría, impidiendo que la red pueda aprender su distribución. Para solucionar este problema, en [12] se sugiere el uso de una función de coste sensitiva o disminuir las muestras negativas.

Con estas indicaciones, hemos modificado la función de coste típica del clasificador binario, la *binary cross-entropy*, añadiendo un peso w al factor utilizado cuando aparece una clase positiva, Eq. 3.2 , tal y como se describe en [13]. Teóricamente, este peso debería ser la proporción de malos respecto a buenos, alrededor de 35, de forma que se igualen los costes, pero con el objetivo de explorar más posibilidades se realizaron distintas pruebas con $w \in [30, 40]$. También hemos implementado un generador de *batches* que disminuye las muestras negativas para conseguir que en cada iteración se saquen datos con la proporción de puntos malos y buenos deseada. Este generador no se aplica en validación y test ya que para que las métricas sean fiables estos conjuntos deben representar la naturaleza del problema.

$$l(\hat{y}, y) = -\frac{1}{N} \sum_{i=1}^{N} w \cdot y_i \cdot ln(\hat{y}_i) + (1 - y_i) ln(1 - \hat{y}_i) \tag{3.2}$$

Una de las principales dificultades que presenta el diseño de la red neuronal es la selección de sus hiperparámetros. La cantidad de posibles elecciones es inmensamente grande por lo que no se pueden explorar todas las combinaciones. Por ello, en este trabajo hemos seleccionado un subconjunto de las posibilidades (reflejado en la tabla 3.1) y hemos realizado una búsqueda del modelo óptimo mediante una exploración aleatoria ya que, según [14], esta ofrece mejores resultados que un muestreo en cuadrícula en el que es posible que zonas interesantes del espacio de búsqueda queden obviadas .

3.2.1. Resultados

En la selección del modelo se han escogido aleatoriamente 200 combinaciones de las opciones mostradas en la tabla 3.1 y se ha entrenado el modelo durante dos épocas (momento en el que empíricamente se comprobó que el coste y el entrenamiento se quedaba estancado). El optimizador SGD (*Stochastic Gradient Descent*) utiliza un *learning rate* de 0.01, el de Adam de 0.001, $\beta_1 = 0.9$ y $\beta_2 = 0.99$ (los recomendados en [15]). Todas las neuronas tienen activación ReLU, que ha demostrado ser más rápida en el aprendizaje que otras activaciones [16], a excepción de la última, que tiene una sigmoide. Ĺa corrección de la función de coste con w nunca se realiza a la vez que controlar la proporción de malos en cada *batch*, pues esto desequilibraría el entrenamiento a favor de los puntos buenos.

La tabla 3.2 recoge las configuraciones y los resultados de los tres mejores modelos obtenidos según el F_1 score. Aunque son modelos distintos es curioso observar como comparten muchas semejanzas entre ellos: número de capas, tamaño del *batch*, tipo de inicialización, optimizador y normalización. Además, los tres modelos han hecho frente al desequilibrio utilizando *batches* con una proporción controlada de puntos malos en lugar de modificando la función de coste. Que los tres mejores modelos encontrados sean tan similares y hayan sido encontrados haciendo una búsqueda aleatoria nos hace

Hiperparámetro	Opciones
Nº de capas ocultas	$1, 2, 3, 4, 5$
Nº de neuronas por capa	$16, 32, 64, 128, 256, 512, 1024$
Tamaño *batch*	$32, 64$
Inicialización de pesos	He *et al.*, Unifome aleatoria
Optimizador	SGD, Adam
Batch Normalization	Sí, No
Proporción malos/buenos	$[0.5, 1.5]$
Peso w en el coste	$[30, 40]$

Tabla 3.1: Espacio de hiperparámetros utilizado en la búsqueda del mejor modelo de perceptrón multicapa.

	Modelo 1	Modelo 2	Modelo 3
Capa 1	512	512	1024
Capa 2	256	256	128
Capa 3	256	128	64
Capa 4	16	16	16
Capa 5	-	-	-
Tamaño *batch*	32	32	32
Inicialización de pesos	He *et al.*	He *et al.*	He *et al.*
Optimizador	Adam	Adam	Adam
Batch Normalization	Sí	Sí	Sí
Proporción malos/buenos	1.04	0.96	0.91
Peso w en el coste	-	-	-
F_1 score	0.0966	0.0956	0.0942
Precision	6.15 %	5.53 %	6.35 %
Recall	22.43 %	35.22 %	18.25 %

Tabla 3.2: Configuración de los tres mejores modelos obtenidos tras la búsqueda aleatoria.

suponer esta configuración es cercana a la del óptimo del espacio de búsqueda.

Incluso en el mejor modelo obtenido (Fig 3.4), las métricas demuestran que este sistema no aportan ninguna ganancia significativa. La precisión es demasiado baja sobre todo contando con que solo se ha encontrado el 22 % de los puntos buenos. Si, por ejemplo, se considerase un clasificador aleatorio, que diga con un 50 % de probabilidades que un punto es bueno, tendrá una precisión del 2.8 % y un *recall* del 50 %. Unos resultados algo peores pero comparables.

3.3. Diseño de una Red Convolucional

En vista de los resultados del apartado anterior se podría pensar que existe un subajuste causado por la utilización de unos modelos demasiado sencillos. Para descartar esta hipótesis, hemos diseñado modelos más complejos utilizando redes convolucionales, un tipo especial de red en el que las neuronas aprovechan la estructura original de los datos de entrada, lo que las hace muy provechosas para el aprendizaje sobre imágenes, o en nuestro caso, sobre parches.

El diseño de nuestra arquitectura está fuertemente basado en el de la red MatchNet [17], un sistema para el emparejamiento de parches. En él, una red convolucional siamesa extrae un vector de cada parche y luego un perceptrón multicapa decide si se trata del mismo punto. Para nuestro propósito

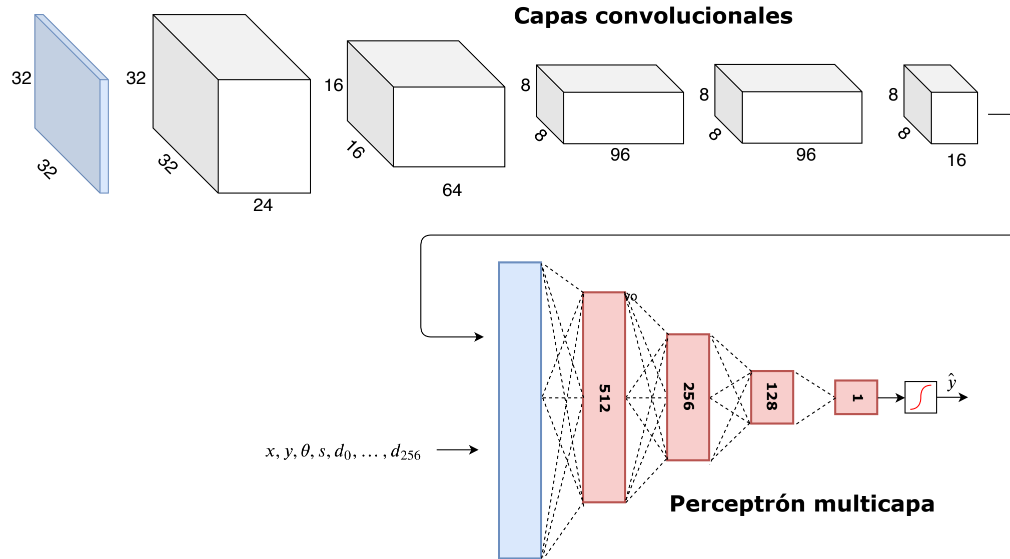


Figura 3.5: Esquema de la mejor arquitectura obtenida utilizando redes convolucionales.

nos ha parecido interesante cómo la red convolucional puede resumir su aprendizaje en un vector que luego es usado por un perceptrón. Nosotros hemos diseñado una red convolucional similar, pero al extraer el vector a partir del parche, hemos concatenado el resto de dimensiones disponibles (posición orientación,...), de forma que la predicción este basada en toda la información disponible, Fig 3.5.

Para seleccionar el modelo se llevó a cabo de nuevo una exploración del espacio de hiperparámetros de forma aleatoria. Sin embargo, en esta ocasión, no se han tenido en cuenta tantas posibilidades ya que los resultados del apartado anterior nos han permitido fijar con seguridad algunas configuraciones (inicialización de He, optimizador de Adam, tamaño de *batch* 32 y *batch normalization.*). Algunas de las características de MatchNet también han sido fijadas (número de capas convolucionales 4 y kernels de cada capa: 7,5,3,3,3).

Concretamente, se ha explorado acerca del tamaño de los filtros de cada capa convolucional, $[16, 32, 64, 96, 128]$, el número de capas densas, $[1, 2, 3, 4]$, proporción de malos en cada iteración, $[0.8, 1.2]$ y si se concatenaban las dimensiones que no son parche (posición, orientación,...) antes de pasar al perceptrón o se prescindía de esta información.

3.3.1. Resultados

Al haber menos hiperparámetros a escoger y tener entrenamientos más costosos (es un modelo bastante más complejo que el del apartado anterior), en esta ocasión solo se realizaron 50 ejecuciones aleatorias.

Tras ellas, se obtuvo que el mejor modelo es el mostrado en la figura 3.5. En este modelo se concatena la salida convolucional con el resto de dimensiones y de nuevo se vuelve a utilizar una proporción de puntos buenos y malos en cada iteración cercana a uno, 1.12, obteniendo una precisión de 6.32 % y un *recall* del 29.15 %. Estos resultados son ligeramente superiores a los del apartado anterior pero siguen siendo demasiado bajos.

3.4. Discusión

El hecho de que ningún modelo diseñado y probado haya conseguido ajustarse al conjunto de datos sugiere que un aprendizaje de la robustez del los puntos ORB en SLAM no es viable. De hecho, las funciones aprendidas por las redes no han conseguido tener un sobreajuste a los datos de entrenamiento, un buen indicador de que la distribución podría ser considerada aleatoria.

Los descriptores ORB no parecen contener información más allá de la que permite el emparejamiento, de forma que no se puede extraer conocimiento de ellos, posiblemente debido a que su naturaleza ingenieril los concibió con la única misión del emparejamiento. Aunque en la creación del dataset se vio como los puntos se concentraban en algunas zonas, la densidad no superaba el 8 % por lo que la posibilidad de acertar aun sabiendo que el punto está en una zona favorable es mínima. El uso de parches ha obtenido los mejores resultados. Sin embargo, las predicciones conseguidas distan mucho de poder ser consideradas útiles en una aplicación real.

En vista de estos resultados se ha concluido que los factores que hacen que un punto ORB sea útil no pueden ser predichos, al menos con la información considerada. Lo provechosa que pueda ser una *feature* de una imagen dependerá del momento en el que sea extraída, y de un gran número de circunstancias que acompañen al sistema en ese instante, pero desde un punto de vista del aprendizaje automático, se ha demostrado que los datos no se rigen por una distribución subyacente, y no pueden ser explicados a partir de los datos del punto, porque a efectos prácticos, su utilidad es aleatoria.

Por estas razones no es posible la inclusión del componente de inteligencia como una capa por encima del extractor y se considera que es necesario prescindir del módulo de extracción original para poder incluir un aprendizaje automático satisfactoriamente.

Capítulo 4

Aprendizaje de las Features

Tal y como se ha concluido en el capítulo anterior, para incluir un aprendizaje del entorno en el sistema es necesario prescindir completamente de la extracción ORB. Por ello, en esta segunda aproximación el módulo de ORB-SLAM2 para detectar puntos de interés ha sido eliminado y sustituido por una red neuronal que marca las zonas de interés de una imagen. Al no contar ya con el extractor ORB no se tienen sus descriptores para realizar su emparejamiento, , y por ello, el método de seguimiento de estas zonas también ha sido sustituido.

Diseñar una red que aprenda la extracción de *features* de un modo *end-to-end* requiere concebir una función de coste completamente derivable que sea capaz de evaluar cuán óptimas son esas *features* para el sistema real. Es difícil definir qué hace a un punto bueno o malo, y normalmente no es posible saberlo hasta que el sistema real sea capaz de utilizarlo. Por consiguiente, se ha descartado la creación de una función con esas características y se ha decidido superar este problema utilizando una red que resuelva un problema similar. Esta solución se basa en la idea intuitiva de que si la red es capaz de resolver un problema similar que requiera un aprendizaje de la geometría del entorno, internamente estará aprendiendo que zonas le son útiles para resolver ese problema, estará fijándose en *features* en sus capas intermedias de forma que las últimas capas de la red las utilicen para sacar el resultado final.

4.1. La red SfMLearner

El sistema SfMLearner presentado por Zhou *et al.* [3] es capaz de aprender la posición relativa entre cámaras y la profundidad de una escena observando una secuencia corta de imágenes, Fig 4.1. Un trabajo similar es realizado por Godard *et al.* [18] utilizando una única imagen como fuente para predecir la profundidad. Para este trabajo hemos decidido utilizar la primera como sistema base ya que el hecho de utilizar una única imagen, como en la segunda, sugiere que no se está aprendiendo a través del movimiento de los objetos en una secuencia si no que se está 'memorizando' la estructura típica de un entorno urbano. En contraste, en la primera red suponemos que se está haciendo un aprendizaje de las consecuencias del movimiento de los objetos, aprendiendo a utilizar los efectos del paralaje en la predicción de la posición de las cámaras.

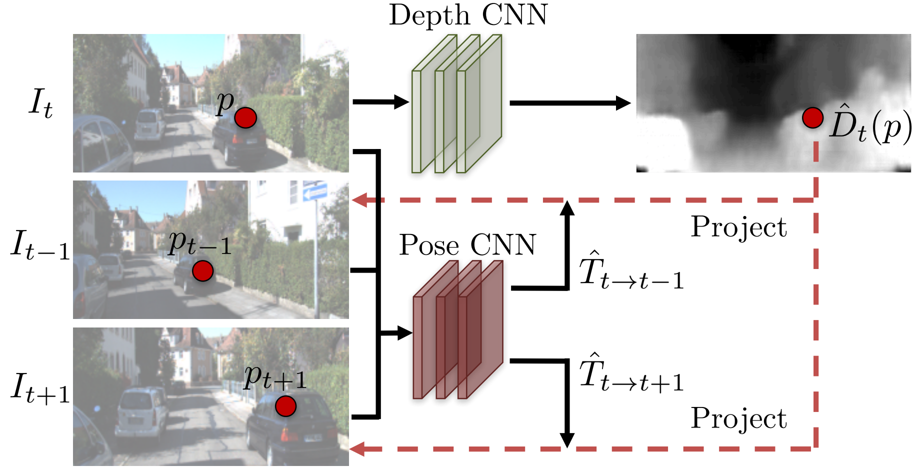


Figura 4.1: Funcionamiento del SfMLearner, una red convolucional predice las transformaciones entre las distintas imágenes, otra red predice la profundidad. Figura extraída de [3].

La red SfMLearner es completamente convolucional y utiliza un método de aprendizaje autosupervisado, es decir, no requiere de *ground truth* para ser entrenada, siendo la red misma la que mide el error que comete en cada iteración. El sistema tiene dos componentes distintas: una red convolucional predice la pose relativa de cada imagen de la secuencia respecto de la imagen objetivo (la imagen central de dicha secuencia) y otra red, también convolucional, deduce el mapa de profundidad de la escena a partir del fotograma objetivo.

Para medir el error cometido, la función de coste a optimizar está basada en la síntesis de nuevas imágenes. De forma más concreta, en la red se tiene una imagen objetivo, I_t, y dos *source*, I_s, (la imagen anterior, I_{t-1} y la siguiente I_{t+1}) y la red predice las matrices transformación de la cámara objetivo a cada *source*, $\hat{T}_{t\to t-1}$ y $\hat{T}_{t\to t+1}$ y el mapa de profundidad de la escena $\hat{D}_t$ (Fig. 4.1). Utilizando esta información se sintetizan las imágenes *source* desde el punto de vista de la objetivo, $\hat{I}_s$, y se calcula el coste comparando la reconstrucción con la imagen original. Para esta reconstrucción se muestrean las coordenadas de los píxeles de I_t en I_s de acuerdo a la ecuación 4.1, donde p_t es un píxel de I_t, p_s es el correspondiente píxel de $\hat{I}_s$ y K es la matriz intrínseca de la cámara.

$$p_s \sim K\hat{T}_{t\to s}\hat{D}_t(p_t)K^{-1}p_t \tag{4.1}$$

Los autores de esta red mantienen un repositorio con la implementación del sistema en TensorFlow en Python. Además, proporcionan un modelo ya entrenado en las secuencias 00-08 del KITTI dataset. Aunque en la explicación original del funcionamiento se utilizaban secuencias de 3 imágenes, el modelo publicado fue entrenado con secuencias de tamaño 5 en cada iteración, lo que según los autores mejoraba el rendimiento en este dataset.

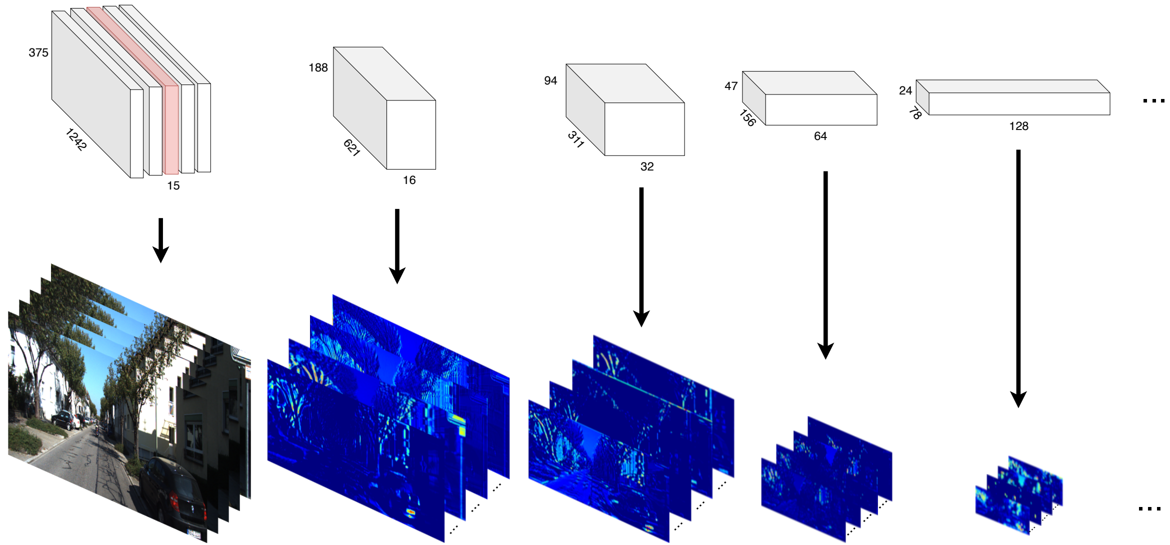


Figura 4.2: Arriba un esquema de la estructura de la red convolucional, con las dimensiones de las salidas de las primeras capas. Abajo algunos de los canales de las respuestas de estas capas. Las primeras capas contienen más información espacial pero menor procesamiento de la red, lo contrario sucede con las últimas, las de la derecha.

4.2. Extracción de Features

Aunque la red SfMLearner tiene dos redes neuronales, predicción y pose, para la extracción de puntos de interés nos hemos centrado en la primera, utilizando el modelo entrenado publicado por los autores. Creemos que esta red es la que más ha podido aprender acerca de los efectos del paralaje, y la que podría contener la información más relevante para nuestro propósito.

4.2.1. Análisis de las capas intermedias

Con el fin de comprobar la hipótesis propuesta de que al aprender una tarea más compleja la red realiza internamente una detección de las zonas de interés hemos visualizado las salidas de las capas convolucionales intermedias. Es difícil decidir cuáles de estas capas pueden ser más útiles para marcar las *features*, las primeras convoluciones tienen mucha información espacial pero la red ha añadido poca parte de su aprendizaje (una única capa convolucional podría ser equivalente a realizar un trabajo similar con un extractor ingenieril), sin embargo, si se mira a las últimas capas, cada valor obtenido proviene de zonas muy amplias de la imagen original, se ha reducido la nitidez, aunque aumentado el procesamiento aportado por la red, Fig. 4.2.

Tras visualizar las distintas salidas de las capas intermedias hemos concluido que el segundo nivel posee el mejor equilibrio entre aportación de la red y mantenimiento de la información espacial. Esta capa tiene 32 filtros distintos, obteniendo una salida de tamaño $94 \times 311 \times 32$. Cada uno de esos filtros se activa ante distintas situaciones, algunas más favorables que otras. Por ejemplo, hay un filtro que produce los valores más altos en la zona que representan el cielo, algo totalmente inútil, mientras otros se fijan en las guías de la carretera o las grietas del suelo, lugares que tienen gran interés para estimar el movimiento de la cámara. El hecho de introducir una secuencia de imágenes como entrada hace que

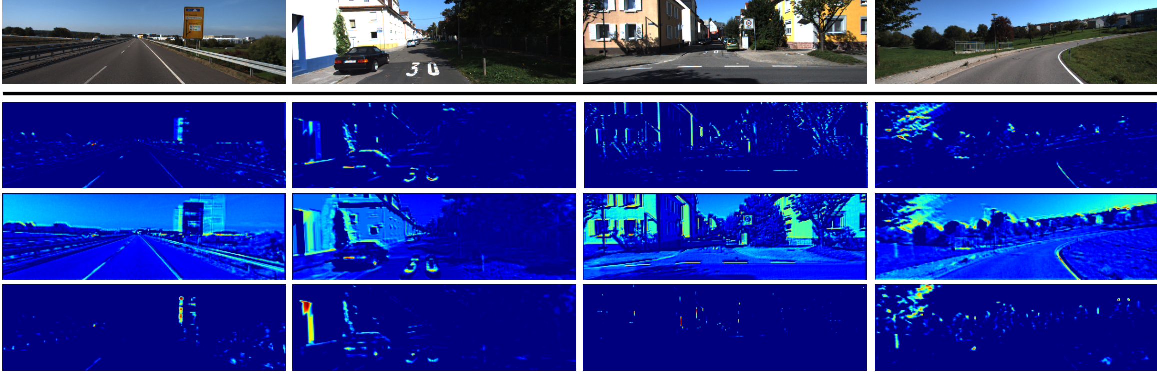


Figura 4.3: Respuesta de los canales 11,12 y 30 de la segunda capa convolucional para algunas imágenes. En la primera columna tercera fila se observa cómo el cartel de la autovía parece activarse varias veces a distintas intensidades. Las marcas del suelo reciben valores altos en la respuesta de la red.

algunas respuestas mezclen información de los distintos fotogramas, Fig. 4.3, pero creemos que este fenómeno no añade mucho ruido porque las respuestas que provienen de las imagen objetivos tienen valores más altos que el resto.

En general, los distintos canales de esta red se fijan en numerosas zonas de interés, por lo que si se tiene en cuenta a todos ellos, parece haber información suficiente como para que esta capa pueda ser utilizada como fuente de puntos de interés.

4.2.2. Elección de Features

La respuesta de la capa elegida produce unos valores altos en lo que se pueden consideran zonas de interés. Sin embargo, estos valores no representan localizaciones exactas sino que indican que esa zona es susceptible de contener puntos interesantes para ser rastreados. El siguiente paso en la construcción de un extractor de *features* requiere convertir estas zonas en localizaciones exactas.

La solución intuitiva más obvia es obtener los máximos locales de la respuesta, es decir, aquellos puntos que tengan un valor más alto que todos sus vecinos en un radio de tamaño r. Esta solución presenta un problema con las zonas poco excitadas ya que un valor muy pequeño rodeado de ceros es un máximo local, pero seguramente no tenga gran interés. Para superar esto sólo se cogen los k mejores valores de cada uno de los 32 filtros, obteniendo un máximo de $32 \times k$ respuestas. La elección del parámetro del tamaño del radio, r, conlleva un consenso entre la distribución de los puntos extraídos a lo largo de la imagen y su precisión. Escoger un valor de r muy alto hará que los puntos estén distribuidos en toda la imagen ya que una sola zona de interés no concentrará muchos de ellos, Fig. 4.4c. Esto tiene el riesgo de que si en una zona sólo se extrae un punto su utilidad depende de que justo ese punto pueda ser rastreado correctamente. Si al contrario se escoge un r bajo, es probable que unas pocas zonas de interés concentren todos los puntos extraídos mientras el resto de la imagen permanece desierta, Fig. 4.4a. La ventaja de esto será qué es más probable alguno de los puntos pertenecientes a esas zonas de interés acabe siendo rastreado correctamente. Empíricamente hemos fijado los parámetros a $r = 11$ y $k = 15$.

(a) $r = 3, k = 15$

(b) $r = 11, k = 15$

(c) $r = 31, k = 15$

Figura 4.4: Extracción de *features* con distintos valores para los parámetros. Al utilizar un radio pequeño los puntos se concentran en las mismas zonas. Con un radio más grande los puntos se distribuyen por toda la imagen.

4.2.3. Integración de TensorFlow en C++

Para poder obtener los puntos de interés extraídos de la red en ORB-SLAM2 es necesario ejecutar el modelo entrenado de TensorFlow (escrito y preparado en Python) en C++. Aunque existe una API oficial de TensorFlow para C++, su uso no esta preparado para integrarse sencillamente con proyectos ya existentes, ya que requiere recompilar todo el código fuente de TensorFlow y utilizar una herramienta de compilación específica, Bazel, que no está preparada para integrarse con CMake, la utilizada por ORB-SLAM2. Esto, junto con la falta de documentación hicieron no viable la integración mediante esta vía.

Otra opción podría ser realizar llamadas foráneas desde C++ a Python, pero este método requiere un gran *overhead* tanto de ejecución como de diseño al tener que realizar continuamente conversiones entre tipos de datos originales y Numpy *arrays*. Además este método es muy *ad hoc*, lo que hace difícil su modificación o escalado.

Por suerte, TensorFlow también ofrece una API en C que no requiere compilar todo el código fuente. Esta API no soporta tipos genéricos (todos los tensores son **(void)*) y es el usuario el encargado de reservar y liberar la memoria dinámica utilizada por la red, lo que es propenso a errores y dificulta el diseño y mantenimiento de los programas. En este trabajo hemos creado una interfaz a esta API para simplificar el proceso de inferencia desde C++. Esta interfaz permite elegir que operaciones desean ser ejecutadas (por su nombre) y suministrar y extraer los datos de la red utilizando vectores tipados de C++, lo que hace que los cambios a realizar si se cambia el modelo sean mínimos y sencillos.

Esta nueva interfaz ha sido publicada (https://github.com/serizba/cppflow) de forma abierta en GitHub para que futuros proyectos puedan realizar la integración de forma sencilla e intuitiva.

4.3. Seguimiento de las Features

Para rastrear los puntos de interés extraídos por la red neuronal entre sucesivas imágenes se ha implementado un *tracker*, en vez de utilizar descriptores, ahora cada punto de una imagen se buscará en otra mirando en los alrededores de la localización esperada.

4.3.1. Seguimiento por correlación

El primer *tracker* implementado ha sido el seguimiento por correlación normalizada. En este algoritmo se compara la similitud del parche alrededor de un punto con una ventana de búsqueda. Para ello se va desplazando el parche, T, de píxel en píxel por toda la ventana, y en cada localización, I, se calcula la correlación según la fórmula 4.2 [19]. El punto de la ventana de búsqueda con un valor más alto indicará el posible emparejamiento, siempre que se supere un cierto umbral, Fig 4.2.

$$\rho = \frac{\sum_{ij}(I_{ij} - \overline{I}_{ij})(T_{ij} - \overline{T}_{ij})}{\sqrt{\sum_{ij}\left(I_{ij} - \overline{I}_{ij}\right)^2}\sqrt{\sum_{ij}\left(T_{ij} - \overline{T}_{ij}\right)^2}} \tag{4.2}$$

Figura 4.5: Ejemplo de seguimiento por correlación. El cuadrado rojo de la imagen de la izquierda marca la plantilla a buscar. La imagen del centro es la ventana de búsqueda, siendo el cuadrado en azul la localización emparejada por el algoritmo. El mapa de calor de la derecha muestra el valor de la correlación en cada punto de la ventana de búsqueda, siendo el valor más alto el que corresponde a las coordenadas del cuadrado azul.

Aunque se trata de un algoritmo bastante rápido, el tiempo de ejecución aumenta considerablemente en función del tamaño de la zona donde se va a buscar la plantilla. Esto es especialmente grave cuando no se tiene una estimación inicial de dónde se va a encontrar el punto (no se puede proyectar) ya que la ventana de búsqueda deberá hacerse excesivamente amplia para aumentar las posibilidades de encontrarlo.

Además, este seguimiento es muy sensible a cambios de tamaño de los objetos, y si se desea añadir invarianza será necesario realizar la búsqueda en distintas escalas de la imagen, lo que también afectará a la velocidad de ejecución.

Estos problemas hacen que no se pueda ejecutar este *tracker* en tiempo real obteniendo buenos resultados.

4.3.2. Seguimiento por Lucas-Kanade

El segundo *tracker* implementado utiliza el método Lucas-Kanade [20], que supera las adversidades del algoritmo anterior efectuando una búsqueda de los puntos de forma más inteligente. Este método hace uso de los gradientes en el punto a buscar para saber en que dirección es más probable encontrar la correspondencia en la otra imagen.

Concretamente, dado un punto y el parche a su alrededor, se intenta buscar el vector (u, v) que determine la dirección en la que se ha movido el punto de interés y que explique los cambios de iluminación en esa zona entre las dos imágenes. Sean I_x e I_y los gradientes de la imagen en horizontal y vertical, (x, y) un punto del parche e $I_t(x, y)$ la diferencia de luminosidad del punto entre las dos imágenes, se puede formar el sistema de ecuaciones 4.3 (para una ventana de 22×22) según el cual, el vector (u, v) marca la dirección del movimiento de los píxeles [21]. Si se resume el sistema de ecuaciones como en 4.4, donde S es una matriz $21^2 \times 2$ con los gradientes y $\vec{t}$ es el vector con las diferencias de intensidades entre los puntos de tamaño 21^2, se puede resolver el sistema sobredeterminado como en 4.5 utilizando el método de mínimos cuadrados.

Figura 4.6: El método Lucas-Kanade consigue realizar emparejamientos cundo se avanza en línea recta (arriba). Sin embargo, en una curva cerrada (abajo), la mayoría de los emparejamientos son erróneos.

$$I_x(x+\Delta x, y+\Delta y)\cdot u + I_y(x+\Delta x, y+\Delta y)\cdot v = -I_t(x+\Delta x, y+\Delta y)$$
$$\forall \Delta x \in [-10, 9, \cdots, 9, 10], \forall \Delta y \in [-10, 9, \cdots, 9, 10] \tag{4.3}$$

$$S\begin{pmatrix} u \\ v \end{pmatrix} = \vec{t} \tag{4.4}$$

$$\begin{pmatrix} u \\ v \end{pmatrix} = (S^T S)^{-1} S^T \vec{t} \tag{4.5}$$

Esté método encontrará una solución si el flujo de movimiento de los puntos es los suficientemente pequeño y si los gradientes contienen una estructura óptima (sobre una estructura difusa sin textura el gradiente será cero). Aunque en nuestro proyecto este método se va a utilizar entre imágenes cercanas a veces el movimiento de la cámara es demasiado grande como para cumplir con estos requisitos, resultando en que se emparejan muy pocos puntos correctamente.

Para superar este problema hemos utilizado una implementación de Lucas-Kanade que incluye la extensión del método presentada en [20] en la que se realiza una pirámide con las escalas de las imágenes de forma que en la cúspide, con una resolución muy baja, los puntos están cerca y sí se cumplen estos requisitos. Además la construcción de la pirámide hace el algoritmo más robusto a cambios de escala.

Con la implementación piramidal, el sistema ORB-SLAM2 consigue funcionar correctamente en tiempo real siempre que no haya movimientos de cámaras demasiado bruscos. Cuando aparece una curva cerrada el movimiento de los puntos es demasiado grande y el método Lucas-Kanade no consigue converger correctamente, Fig 4.6.

4.3.3. Cómo sobrevivir a las curvas

Ante una curva cerrada, los *MapPoints* sí consiguen ser rastreados correctamente, ya que al conocer su ubicación espacial se pueden proyectar y tener una estimación inicial de dónde encontrarlos. El problema aparece cuando se desea emparejar y triangular nuevos puntos, pues no se conoce su posición tridimensional, y si esta tarea no se consigue realizar, en el momento en el que los *MapPoints* desaparezcan del campo de visión de la cámara el sistema se perderá.

Estas triangulaciones solo se llevan a cabo entre *Keyframes*, por lo que sería conveniente que durante las curvas, y justo antes de que comiencen, se insertasen con mayor frecuencia y así el movimiento de los puntos fuese más suave. Sin embargo, en la práctica, cuando aparece una curva el automóvil suele reducir la velocidad, haciendo que ORB-SLAM2 considere que las imágenes son redundantes y que no se deben insertar *Keyframes*.

Tratando de aliviar este problema hemos modificado las políticas de inserción y borrado de *Keyframes* para que un fotograma pase a considerarse de gran importancia si existe una gran rotación respecto a la última imagen procesada. Para ello se extrae la rotación entre las dos cámaras a partir de sus matrices transformación, Eq. 4.6, y de esa matriz se calcula la rotación en cada eje para evaluar el giro total realizado. Aunque esta mejora ha conseguido aliviar algunas curvas, no es suficiente, pues incluso con fotogramas contiguos, los puntos se siguen desplazando demasiado.

$$R_{c_1 \to c_2} = R_{w \to c_1}^T \cdot R_{w \to c_2} \tag{4.6}$$

Para conseguir solucionar el problema bastaría con tener una leve intuición de dónde se van encontrar los puntos. Por ello, hemos planteado una simplificación que nos permita realizar proyecciones que, aunque no sean certeras, puedan acercar la estimación del punto real y ayudar al método Lucas-Kanade. Esta simplificación consiste en la suposición de que no ha existido traslación entre las dos cámaras y su movimiento ha sido simplemente en rotación. De esta forma es posible calcular la proyección, pues la posición tridimensional de los puntos solo es necesaria si se quiere tener en cuenta la traslación. Para ello calculamos la homografía que relaciona las dos imágenes y la utilizamos para proyectar los puntos de una imagen en la otra, Eq. 4.7.

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = K \cdot R_{c_1 \to c_2} \cdot K^{-1} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{4.7}$$

A pesar de que se trata de una simplificación, las predicciones son lo suficientemente acertadas cómo para permitir que el método de Lucas-Kanade las pueda corregir y obtener emparejamientos correctos, Fig. 4.7.

Con esta mejora el sistema es capaz de funcionar correctamente a tiempo real en el dataset urbano de KITTI.

Figura 4.7: Al proyectar los puntos utilizando la homografía la mayoría de los puntos se empareja correctamente tanto al avanzar en línea recta (arriba) como en una curva cerrada (abajo).

4.4. Resultados

Como nuestro sistema utiliza la red SfMLearner, que ha sido entrenada en las secuencias 00-08 del KITTI, la evaluación del programa se ha realizado sobre las dos únicas secuencias nunca vistas por la red, la 09 y la 10, Fig 4.8. Dado que en nuestro sistema ya no se realiza el cerrado de bucles, las comparaciones han sido realizadas con una versión de ORB-SLAM2 en las mismas condiciones, con el hilo *Loop Closing* desactivado.

Para intentar diferenciar qué resultados son consecuencia del nuevo seguimiento por Lucas-Kanade y cuáles son debido a la extracción de *features* utilizando la red, en las comparaciones hemos incluido una versión de nuestro sistema que utiliza el extractor de Shi Tomasi [22] en lugar de la red.

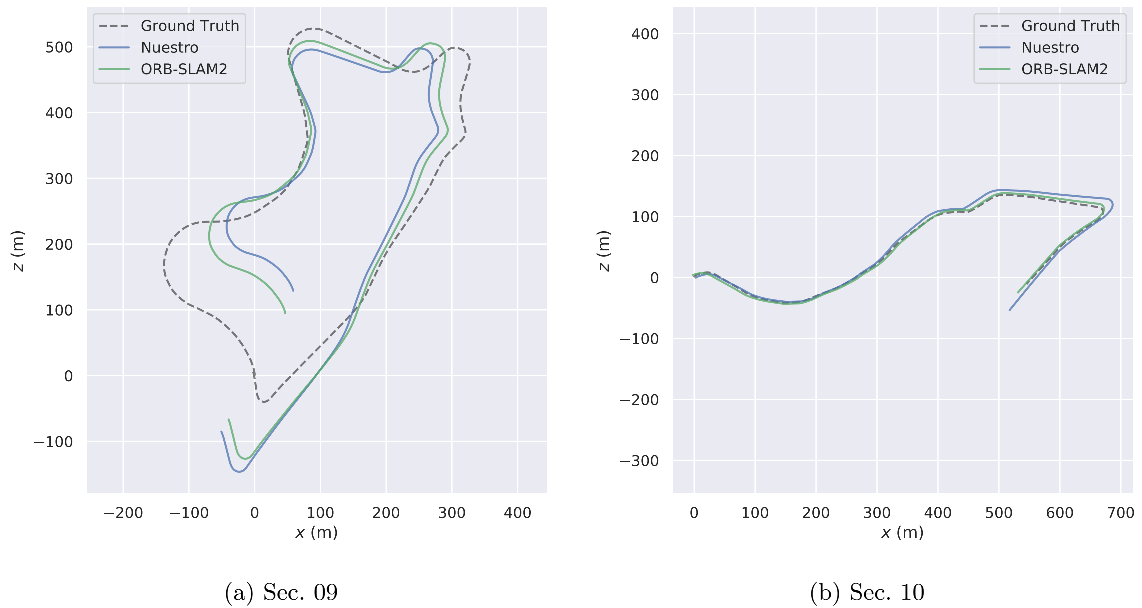
Intentando conseguir resultados fiables, cada secuencia ha sido ejecutada 5 veces por sistema. La media de estas ejecuciones, reflejada en la tabla 4.1, muestra cómo el ORB-SLAM2 original obtiene los mejores resultados por una estrecha ventaja. Comparando nuestra versión con red y sin red se tiene que utilizándola se consiguen mejores resultados, lo que nos indica y demuestra que las salidas de la red son realmente interesantes.

Atendiendo a uno de nuestros objetivos iniciales, intentar aprovechar una mayor cantidad de puntos extraídos, sí que se ha conseguido una mejora significativa. En la figura 4.9 se observa cómo los puntos aprovechados han pasado de ser una escasa minoría (Fig 2.4) a ser la mayor parte. Ahora el 56.4 % de los puntos extraídos son útiles.

El tiempo medio de procesamiento de cada imagen de nuestro sistema (0.0489 seg.) es superior al de ORB-SLAM2 (0.023 seg.), lo que es en su mayor parte debido al nuevo *tracker*.

	RMSE Traslación Sec. 09 (m)	RMSE Rotación Sec. 09 (deg)	RMSE Traslación Sec. 10 (m)	RMSE Rotación Sec. 10 (deg)
ORB-SLAM2	41.187	0.083	6.923	0.025
Nuestro con CNN	73.898	0.138	11.953	0.032
Nuestro con Shi Tomasi	93.042	0.224	17.050	0.053

Tabla 4.1: Comparación de los RMSE del error de trayectoria absoluta (ATE) en traslación y rotación en las secuencias 09 y 10 del KITTI dataset.



(a) Sec. 09

(b) Sec. 10

Figura 4.8: Trayectoria de los *Keyframes* de ORB-SLAM2 y nuestro sistema en las secuencias 09 y 10 del KITTI.

Figura 4.9: Con nuestro sistema hay mas puntos extraídos buenos (verdes) que malos (rojo).

Capítulo 5

Conclusiones

Los resultados obtenidos en la primera parte de este proyecto nos han permitido demostrar que la utilidad de un punto de interés ORB no puede ser predicha, al menos con la información considerada. Haciendo una búsqueda de diferentes modelos de red neuronal con mayor o menor complejidad nos hemos encontrado con que ningún diseño es capaz de aprender la distribución de los datos, ni siquiera de los de entrenamiento, una certera señal de que tal distribución no tiene una estructura predecible.

Por otro lado, hemos conseguido utilizar una red convolucional como fuente de información para implementar un extractor de *features* que ha resultado ser mejor que el establecido algoritmo de Shi-Tomasi [22] para nuestro problema en concreto.

Este extractor, combinado con un nuevo sistema de seguimiento de los puntos de interés utilizando una implementación del método Lucas-Kanade, ha conseguido que el porcentaje de puntos aprovechables por el sistema aumente de forma radical: de una tasa del 3 % en el ORB-SLAM2 original al 56 % con nuestro sistema. La validación de nuestro sistema en dos secuencias nunca vistas por la red ha demostrado que el extractor es capaz de generalizar a nuevas situaciones al haber conseguido una precisión similar, aunque inferior, a la obtenida por ORB-SLAM2

5.1. Trabajo Futuro

Incluir redes neuronales en ORB-SLAM2 abre un gran abanico de posibilidades. Aunque este trabajo se ha centrado en la extracción de *features*, hay numerosas formas en las que un sistema de SLAM podría complementarse o ayudarse de una red neuronal.

En este proyecto se ha tenido que desactivar el cerrado de bucles como consecuencia de no utilizar puntos ORB. Un futuro trabajo podría tratar de utilizar aprendizaje profundo para detectar cuando se esta pasando por una zona ya visitada de forma robusta y precisa.

Otra interesante línea de investigación podría emplear redes neuronales de predicción de profundidad para ayudar al sistema con una estimación inicial de la estructura de la escena, lo que podría ayudar a reducir o evitar la deriva de escala.

Bibliografía

[1] R. Mur-Artal y J. D. Tardós, “ORB-SLAM2: An open-source slam system for monocular, stereo, and rgb-d cameras”, *IEEE Transactions on Robotics*, vol. 33, n.º 5, págs. 1255-1262, 2017.

[2] R. H. A. Zisserman, “Multiple view geometry in computer vision”, 2004.

[3] T. Zhou, M. Brown, N. Snavely y D. G. Lowe, “Unsupervised learning of depth and ego-motion from video”, en *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, págs. 1851-1858.

[4] Wikipedia contributors, *Depth perception — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Depth_perception&oldid=901181354`, [Online; accessed 19-June-2019], 2019.

[5] B. Triggs, P. F. McLauchlan, R. I. Hartley y A. W. Fitzgibbon, “Bundle adjustment—a modern synthesis”, en *International workshop on vision algorithms*, Springer, 1999, págs. 298-372.

[6] R. Mur-Artal, J. M. M. Montiel y J. D. Tardos, “ORB-SLAM: A versatile and accurate monocular slam system”, *IEEE transactions on robotics*, vol. 31, n.º 5, págs. 1147-1163, 2015.

[7] E. Rublee, V. Rabaud, K. Konolige y G. R. Bradski, “ORB: An efficient alternative to SIFT or SURF.”, en *ICCV*, Citeseer, vol. 11, 2011, pág. 2.

[8] E. Rosten, R. Porter y T. Drummond, “Faster and better: A machine learning approach to corner detection”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, n.º 1, págs. 105-119, ene. de 2010, ISSN: 0162-8828. DOI: `10.1109/TPAMI.2008.275`.

[9] M. Calonder, V. Lepetit, C. Strecha y P. Fua, “BRIEF: Binary robust independent elementary features”, en *European conference on computer vision*, Springer, 2010, págs. 778-792.

[10] C. G. Harris, M. Stephens y col., “A combined corner and edge detector.”, en *Alvey vision conference*, Citeseer, vol. 15, 1988, págs. 10-5244.

[11] A. Geiger, P. Lenz y R. Urtasun, “Are we ready for autonomous driving? the KITTI vision benchmark suite”, en *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[12] M. Buda, A. Maki y M. A. Mazurowski, “A systematic study of the class imbalance problem in convolutional neural networks”, *Neural Networks*, vol. 106, págs. 249-259, 2018.

[13] M. Kukar, I. Kononenko y col., “Cost-sensitive learning with neural networks.”, en *ECAI*, 1998, págs. 445-449.

[14] J. Bergstra e Y. Bengio, “Random search for hyper-parameter optimization”, *Journal of Machine Learning Research*, vol. 13, n.º Feb, págs. 281-305, 2012.

[15] D. P. Kingma y J. Ba, “Adam: A method for stochastic optimization”, *ArXiv preprint arXiv:1412.6980*, 2014.

[16] A. Krizhevsky, I. Sutskever y G. E. Hinton, “Imagenet classification with deep convolutional neural networks”, en *Advances in neural information processing systems*, 2012, págs. 1097-1105.

[17] X. Han, T. Leung, Y. Jia, R. Sukthankar y A. C. Berg, “Matchnet: Unifying feature and metric learning for patch-based matching”, en *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, págs. 3279-3286.

[18] C. Godard, O. Mac Aodha y G. J. Brostow, “Unsupervised monocular depth estimation with left-right consistency”, en *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, jul. de 2017.

[19] OpenCV. (2019). Template matching, dirección: https://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/template_matching/template_matching.html (visitado 25-06-2019).

[20] J.-Y. Bouguet y col., "Pyramidal implementation of the affine Lucas Kanade feature tracker description of the algorithm", *Intel Corporation*, vol. 5, n.º 1-10, pág. 4, 2001.

[21] R. Rojas, "Lucas-kanade in a nutshell",

[22] Jianbo Shi y Carlo Tomasi, "Good features to track", en *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, jun. de 1994, págs. 593-600. DOI: 10.1109/CVPR.1994.323794.

[23] B. D. Lucas y T. Kanade, "An iterative image registration technique with an application to stereo vision", en *Int. Joint. Conf. on Artificial Intelligence (IJCAI)*, 1981, págs. 674-679.

[24] I. Goodfellow, Y. Bengio y A. Courville, *Deep learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[25] L. Fei-Fei, J. Johnson y S. Yeung, "Cs231n: Convolutional neural networks for visual recognition", http://vision.stanford.edu/teaching/cs231n/, Standford University, 2019.

[26] K. He, X. Zhang, S. Ren y J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", en *Proceedings of the IEEE international conference on computer vision*, 2015, págs. 1026-1034.

[27] S. Ioffe y C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", *ArXiv preprint arXiv:1502.03167*, 2015.

[28] G. Bradski, "The OpenCV Library", *Dr. Dobb's Journal of Software Tools*, 2000.

[29] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu y Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. dirección: https://www.tensorflow.org/.

[30] E. Jones, T. Oliphant, P. Peterson y col., *SciPy: Open source scientific tools for Python*, 2001–. dirección: http://www.scipy.org/.

[31] F. Chollet y col., *Keras*, https://keras.io, 2015.

Anexos

Anexos A

Herramientas Utilizadas

En el desarrollo de este proyecto se han utilizado las siguientes herramientas y tecnologías:

ORB-SLAM2: Sistema de SLAM creado por la Universidad de Zaragoza [1].

OpenCV: Librería de código abierto de visión por computador [28].

Tensorflow: Librería de código abierto para la creación de modelos de aprendizaje automático [29].

Keras: Librería de código abierto basada en TensorFlow para el diseño y entrenamiento de redes neuronales [31].

SciPy: Ecosistema de código abierto con herramientas y algoritmos para uso científico o matemático [30].

Git: Software de control de versiones.

LaTeX: Sistema de composición de textos de gran calidad.

Anexos B

Gestión del Proyecto

Este proyecto se ha realizado dentro de la investigación llevada a cabo con una Beca de Colaboración.

Su dedicación aproximada ha sido de 3 horas diarias (15 horas semanales). La tabla B.1 expone cuántas horas ha requerido cada una de las tareas de este proyecto. La figura B.1 muestra cómo se han distribuido esas tareas a lo largo del curso académico.

Tarea	Horas
Aprendizaje de ORB-SLAM2	25
Generación dataset	40
Aprendizaje redes	25
Implementación redes	90
Extracción features	60
Reimplementación del hilo tracking	135
Memoria	70
Total	445

Tabla B.1: Horas dedicadas a cada tarea de este proyecto.

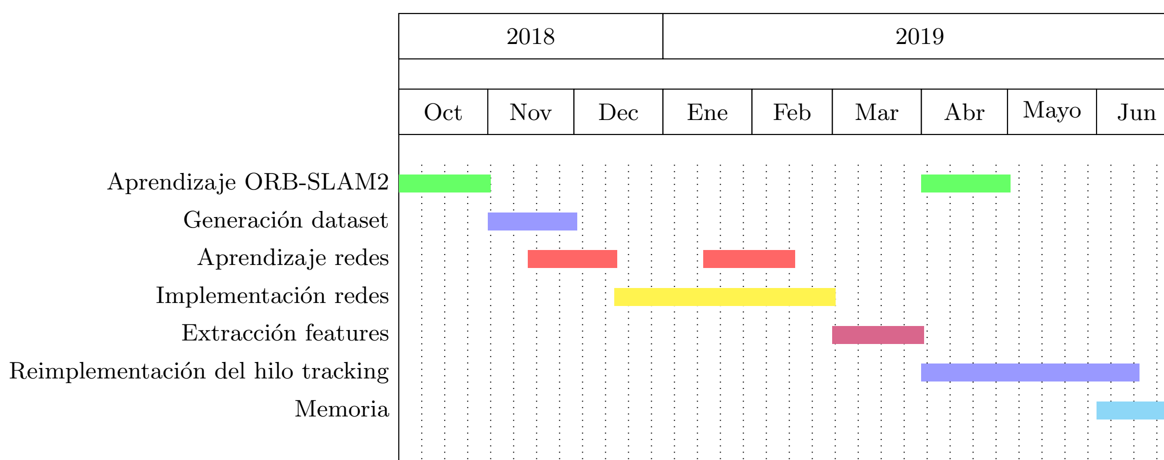


Figura B.1: Diagrama de Gantt para exponer en qué periodos se ha realizado cada tarea.