

UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y
AUTOMÁTICA INDUSTRIAL



“Deep Learning aplicado a la
reconstrucción 3D estereoscópica”

TRABAJO FIN DE GRADO

Junio - 2026

AUTOR: Ángel Mejías Martínez

DIRECTOR: Luis Miguel Jimenez García

ÍNDICE:

1. INTRODUCCIÓN.....	3
1.1. CONTEXTO DEL PROYECTO.....	3
1.2. MOTIVACIÓN.....	3
1.3. OBJETIVOS DEL TRABAJO.....	4
1.4. ESTRUCTURA DE LA MEMORIA.....	4
2. ESTADO DEL ARTE.....	6
2.1. RECONSTRUCCIÓN 3D MEDIANTE VISIÓN POR COMPUTADOR.....	6
2.2. FLUJO GENERAL DEL MÉTODO CLÁSICO DE RECONSTRUCCIÓN 3D ESTÉREO.....	7
2.3. MODELO DE CÁMARA, CALIBRACIÓN Y DISTORSIÓN DE IMAGEN.....	7
2.4. DISPOSICIÓN DE LAS CÁMARAS: EJES PARALELOS Y EJES CONVERGENTES.....	9
2.5. GEOMETRÍA EPIPOLAR: RESTRICCIÓN FUNDAMENTAL DEL PAR ESTÉREO.....	10
2.6. RECTIFICACIÓN EPIPOLAR.....	12
2.7. CORRESPONDENCIA ESTÉREO.....	13
2.7.1. CORRESPONDENCIA DISPERSA BASADA EN CARACTERÍSTICAS.....	14
2.7.2. CORRESPONDENCIA Densa BASADA EN ÁREA.....	16
2.8. MAPA DE DISPARIDAD Y REGULARIZACIÓN.....	17
2.9. TRIANGULACIÓN, PROFUNDIDAD Y NUBE DE PUNTOS.....	17
2.10. LIMITACIONES DEL MÉTODO CLÁSICO.....	20
2.11. APRENDIZAJE PROFUNDO APLICADO A VISIÓN ESTÉREO.....	20
2.12. DATASETS, BENCHMARKS Y MÉTRICAS DE EVALUACIÓN.....	22
2.13. SÍNTESIS DEL ESTADO DEL ARTE Y RELACIÓN CON EL TRABAJO.....	22
3. MATERIALES Y MÉTODOS.....	24
3.1. ENTORNO DE DESARROLLO.....	24
3.2. LIBRERÍAS Y HERRAMIENTAS UTILIZADAS.....	24
3.3. ALMACENAMIENTO Y ORGANIZACIÓN DE ARCHIVOS.....	34
3.4. MÉTODO BASADO EN PSMNET.....	38
3.5. PESOS PREENTRENADOS Y TRANSFERENCIA DE APRENDIZAJE.....	40
3.6. DATASETS UTILIZADOS.....	41
3.7. MÉTRICAS UTILIZADAS.....	43
3.8. ESQUEMA GENERAL DE LA METODOLOGÍA EXPERIMENTAL.....	44

4. RESULTADOS Y DISCUSIÓN.....	46
4.1. MÉTODO CLÁSICO DE COMPARACIÓN.....	46
4.2. FINE-TUNING Y EXPERIMENTOS DE ENTRENAMIENTO.....	47
4.2.1. ITERACIONES DE ENTRENAMIENTO Y ÉPOCAS.....	47
4.2.2. EXPERIMENTO PREVIO CON EARLY STOPPING.....	49
4.2.3. FINE-TUNING EN DIFERENTES CAPAS.....	49
4.3. EVALUACIÓN.....	54
4.3.1. MÉTRICAS PRINCIPALES.....	54
4.3.2. ANÁLISIS DE SOBREAJUSTE.....	58
4.3.3. OTRAS MÉTRICAS.....	59
4.4. COMPARACIÓN DE MAPAS DE DISPARIDAD.....	62
4.5. RECONSTRUCCIÓN 3D.....	70
4.5.1. DESCRIPCIÓN DEL PROCESO DE RECONSTRUCCIÓN 3D.....	70
4.5.2. RESULTADOS VISUALES DE LA RECONSTRUCCIÓN 3D.....	72
4.5.3. CALIBRACIÓN Y RECTIFICACIÓN EPIPOLAR.....	80
5. CONCLUSIONES.....	82
5.1. CONCLUSIONES PRINCIPALES DEL TRABAJO.....	82
5.2. ANÁLISIS DE RESULTADOS Y LIMITACIONES.....	83
5.3. TRABAJOS FUTUROS.....	84
6. BIBLIOGRAFÍA.....	86
7. ANEXOS.....	87
7.1. ANEXO 1: EVALUACIÓN DEL MÉTODO CLÁSICO.....	87
7.2. ANEXO 2: FINE-TUNING DE TODOS LOS EXPERIMENTOS.....	113
7.3. ANEXO 3: EVALUACIÓN DEL FINE-TUNED Y PREENTRENO.....	143
7.4. ANEXO 4: COMPARATIVA MAPAS DE DISPARIDAD.....	177
7.5. ANEXO 5: RECONSTRUCCIÓN 3D.....	208

1. INTRODUCCIÓN

1.1. CONTEXTO DEL PROYECTO

La reconstrucción tridimensional del entorno a partir de imágenes constituye uno de los problemas fundamentales dentro del ámbito de la visión por computador. Su objetivo es obtener información de profundidad y geometría de una escena real a partir de datos bidimensionales, permitiendo generar representaciones espaciales útiles para interpretar el entorno.

Dentro de este campo, la visión estéreo se basa en la adquisición de dos imágenes desde posiciones ligeramente distintas. A partir de estas imágenes es posible calcular la disparidad entre puntos correspondientes y, mediante relaciones geométricas, estimar la distancia de los objetos respecto a la cámara [1].

Este enfoque, inspirado en el sistema visual humano, permite obtener mapas de profundidad densos sin necesidad de sensores activos, lo que lo convierte en una solución de interés en numerosos sistemas de percepción.

1.2. MOTIVACIÓN

La estimación de profundidad mediante visión estéreo se ha abordado tradicionalmente mediante técnicas geométricas y algoritmos de correspondencia. Aunque estos métodos presentan una base matemática sólida, su rendimiento puede verse afectado en escenarios reales por oclusiones, superficies con baja textura, patrones repetitivos o cambios de iluminación, donde la correspondencia entre ambas imágenes resulta ambigua.

En los últimos años, las redes neuronales convolucionales han permitido mejorar la estimación de disparidad al aprender representaciones visuales directamente a partir de los datos. No obstante, estos modelos tampoco eliminan por completo las dificultades propias del problema estéreo, especialmente en zonas ocluidas, regiones con poca información visual o situaciones con cambios importantes entre dominios.

La motivación de este trabajo surge del interés por estudiar la aplicación de redes neuronales convolucionales a la reconstrucción 3D estéreo, evaluando su comportamiento frente a técnicas tradicionales y analizando su capacidad para generar mapas de disparidad útiles en escenas reales.

1.3. OBJETIVOS DEL TRABAJO

El objetivo principal de este Trabajo Fin de Grado es desarrollar e implementar un sistema de reconstrucción 3D basado en aprendizaje profundo, utilizando una arquitectura de red neuronal convolucional para estimar mapas de disparidad a partir de pares de imágenes estéreo. Para alcanzar este objetivo principal se plantean los siguientes objetivos específicos:

- Estudiar los fundamentos de la visión estéreo y de la reconstrucción 3D a partir de mapas de disparidad.
- Revisar las técnicas clásicas y los métodos basados en aprendizaje profundo aplicados a la estimación de disparidad.
- Implementar y evaluar la arquitectura PSMNet utilizando pesos preentrenados.
- Aplicar técnicas de fine-tuning para adaptar el modelo a datasets reales de visión estéreo.
- Evaluar el rendimiento del modelo mediante métricas cuantitativas como EPE, D1-all, AbsRel y delta1.
- Comparar los resultados obtenidos por la red neuronal con un método clásico de correspondencia estéreo.
- Generar visualizaciones de los mapas de disparidad y reconstrucciones tridimensionales que permitan analizar cualitativamente los resultados.

1.4. ESTRUCTURA DE LA MEMORIA

La memoria se organiza en siete capítulos. En primer lugar, la introducción presenta el contexto del proyecto, la motivación, los objetivos y la estructura general del documento. A continuación, el estado del arte revisa las principales técnicas de reconstrucción 3D aplicables al trabajo, desde los métodos clásicos de visión estéreo hasta las aproximaciones basadas en redes neuronales convolucionales.

El capítulo de materiales y métodos describe el entorno de desarrollo, las librerías empleadas, los datasets utilizados, la arquitectura seleccionada. Posteriormente, el capítulo de resultados experimentales recogerá la evaluación cuantitativa y cualitativa de los experimentos realizados: fine-tuning, evaluación final para determinar pesos, comparación entre mapas de disparidad y reconstrucción 3D.

Finalmente, se presentan las conclusiones del trabajo, la bibliografía consultada y los anexos, donde se incluyen los diferentes códigos desarrollados.



2. ESTADO DEL ARTE

2.1. RECONSTRUCCIÓN 3D MEDIANTE VISIÓN POR COMPUTADOR

La reconstrucción tridimensional mediante visión por computador consiste en obtener información espacial de una escena a partir de una o varias imágenes. A diferencia de una imagen convencional, que únicamente representa la escena sobre un plano bidimensional, la reconstrucción 3D busca recuperar la profundidad, la forma y la posición relativa de los objetos presentes en el entorno [2].

Existen distintas formas de estimar la profundidad. Los sensores activos, como LiDAR o cámaras de tiempo de vuelo, emiten una señal y miden su retorno. En cambio, las técnicas pasivas basadas en cámaras convencionales no emiten energía sobre la escena, sino que infieren la geometría a partir de las imágenes captadas. Dentro de estas técnicas pasivas, la visión estéreo es una de las aproximaciones clásicas más utilizadas, ya que permite estimar profundidad a partir de dos imágenes tomadas desde puntos de vista ligeramente distintos.

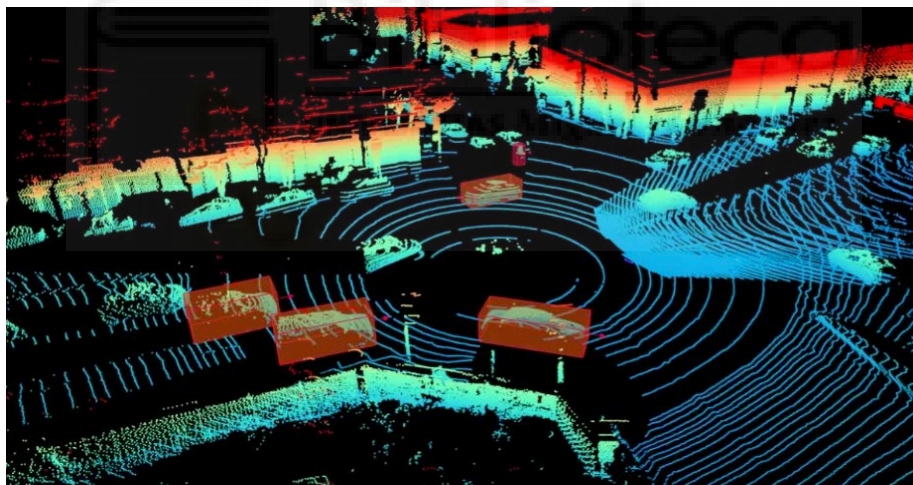


Figura 2.1. Representación tridimensional de un entorno urbano mediante una nube de puntos capturada por un sensor LiDAR.

En un sistema estéreo, la información 3D no aparece directamente en ninguna de las dos imágenes por separado. La profundidad se obtiene comparando ambas vistas, localizando puntos correspondientes y aplicando relaciones geométricas. Por ello, el problema central de la visión estéreo es doble: primero encontrar correspondencias fiables entre las imágenes y después reconstruir la posición espacial de los puntos mediante triangulación.

2.2. FLUJO GENERAL DEL MÉTODO CLÁSICO DE RECONSTRUCCIÓN 3D ESTÉREO

El método clásico o geométrico de reconstrucción estéreo se basa en una cadena de etapas bien definida. Aunque cada implementación puede variar, el proceso completo suele seguir el siguiente orden: captura del par estéreo, calibración de las cámaras, corrección de distorsión, rectificación epipolar, búsqueda de correspondencias, cálculo de disparidad, triangulación y generación de una representación 3D, normalmente en forma de mapa de profundidad o nube de puntos [2].

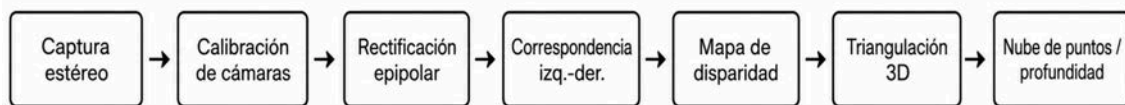


Figura 2.2. Flujo general de una reconstrucción 3D estéreo clásica.

La importancia de este flujo es que cada etapa condiciona a la siguiente. Una calibración imprecisa provoca rectificaciones incorrectas; una rectificación deficiente dificulta la correspondencia; errores en la correspondencia generan disparidades erróneas; y los errores de disparidad se amplifican en la estimación de profundidad, especialmente para puntos lejanos. Por este motivo, los métodos clásicos tienen una base matemática sólida, pero son sensibles a la calidad de las imágenes, a la geometría del sistema estéreo y a la textura de la escena.

2.3. MODELO DE CÁMARA, CALIBRACIÓN Y DISTORSIÓN DE IMAGEN

La calibración de cámara sí pertenece al problema estéreo, aunque también puede estudiarse de forma independiente para una sola cámara. En reconstrucción 3D estéreo es necesaria porque permite conocer cómo se proyectan los puntos del mundo sobre cada imagen y cuál es la relación geométrica entre las dos cámaras [1].

El modelo más habitual es el modelo de cámara pinhole o modelo proyectivo. En este modelo, un punto tridimensional $M = (X, Y, Z)$ se proyecta sobre un punto de imagen $m = (u, v)$. En coordenadas homogéneas, la proyección puede expresarse como:

$$s \cdot [u, v, 1]^T = K \cdot [R|t] \cdot [X, Y, Z, 1]^T$$

donde s es un factor de escala, K es la matriz de parámetros intrínsecos de la cámara, R y t representan la rotación y traslación respecto al sistema de referencia elegido, y $[R \mid t]$ contiene los parámetros extrínsecos.

La matriz intrínseca suele escribirse como:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Los parámetros f_x y f_y representan la distancia focal expresada en píxeles, mientras que (c_x, c_y) es el punto principal de la imagen. Estos parámetros permiten transformar coordenadas normalizadas de cámara en coordenadas de píxel. En un par estéreo, además de calibrar cada cámara por separado, se necesita conocer la transformación rígida entre ambas, normalmente expresada mediante una rotación R y una traslación t . La distancia entre los centros ópticos se denomina línea base o baseline, representada habitualmente como B .

Las lentes reales no cumplen exactamente el modelo pinhole ideal. Por ello, antes de reconstruir en 3D suele ser necesario corregir la distorsión de la imagen. La distorsión radial hace que las líneas rectas puedan aparecer curvadas, especialmente cerca de los bordes de la imagen. La distorsión tangencial aparece cuando la lente no está perfectamente alineada con el plano del sensor. Un modelo habitual corrige las coordenadas normalizadas (x, y) mediante coeficientes radiales k_1, k_2, k_3 y tangenciales p_1, p_2 :

$$r^2 = x^2 + y^2$$

$$x_d = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 xy + p_2(r^2 + 2x^2)$$

$$y_d = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_2 xy + p_1(r^2 + 2y^2)$$

En la práctica, la calibración proporciona los parámetros necesarios para eliminar o reducir esta distorsión. Si no se corrige, dos puntos que deberían estar relacionados por una misma línea epipolar pueden aparecer desplazados de forma no geométrica, lo que aumenta los errores de correspondencia y de triangulación.

Etapa	Qué estima	Para qué sirve en estéreo
Calibración intrínseca	f_x, f_y, c_x, c_y y coeficientes de distorsión	Permite pasar de píxeles a rayos de cámara y corregir deformaciones de la lente.
Calibración extrínseca	Rotación R y traslación t entre cámaras	Define la geometría relativa del par estéreo y la baseline B .
Rectificación	Transformaciones de remapeo de ambas imágenes	Alinea las líneas epipolares para simplificar la correspondencia.

2.4. DISPOSICIÓN DE LAS CÁMARAS: EJES PARALELOS Y EJES CONVERGENTES

La disposición de las cámaras condiciona la geometría del sistema estéreo. Las dos configuraciones más habituales son los sistemas de ejes paralelos o alineados y los sistemas de ejes convergentes.

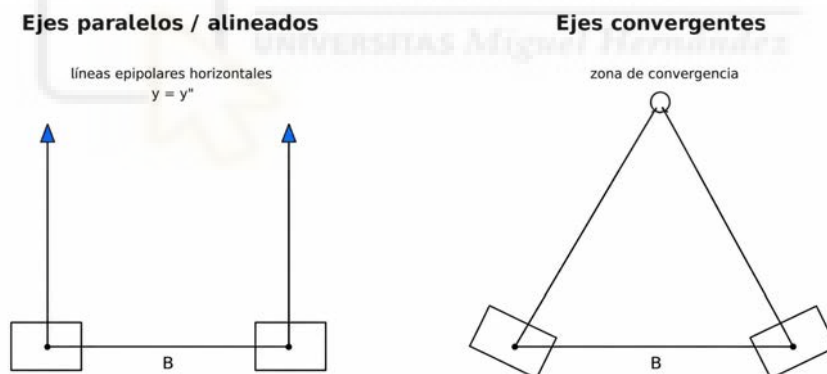


Figura 2.3. Comparación conceptual entre ejes paralelos o alineados y ejes convergentes.

En un sistema de ejes paralelos, los ejes ópticos de ambas cámaras son paralelos y los sensores se encuentran alineados. Si además las cámaras tienen la misma orientación y las imágenes están rectificadas, las líneas epipolares son horizontales. Esto permite buscar la correspondencia de un punto únicamente sobre la misma fila de la otra imagen, reduciendo el problema de búsqueda de dos dimensiones a una dimensión.

En un sistema de ejes convergentes, las cámaras se orientan hacia una zona común de la escena. Esta configuración puede aumentar la región útil observada por ambas cámaras en determinadas condiciones y facilita la construcción física en algunos montajes, pero complica el modelo geométrico porque las líneas epipolares no son, en general, horizontales. En este caso la triangulación suele formularse de forma general mediante matrices de proyección y la correspondencia debe respetar la geometría epipolar.

Configuración	Ventajas	Desventajas
Ejes paralelos / alineados	Modelo de cálculo más simple. En imágenes rectificadas se cumple $y_L = y_R$. La disparidad se interpreta como desplazamiento horizontal.	Puede ser difícil construir un montaje perfectamente alineado. Si la baseline es grande, las imágenes pueden ser muy distintas y aparecer más oclusiones.
Ejes convergentes	Puede aumentar el campo común útil en ciertas escenas. Permite apuntar ambas cámaras a una zona de interés. Puede facilitar la correspondencia si la disparidad aparente se mantiene baja.	Modelo matemático más complejo. Requiere tratar líneas epipolares no necesariamente horizontales o rectificar previamente.

La baseline B también introduce un compromiso importante. Una baseline pequeña genera imágenes muy parecidas y un campo común mayor, pero produce disparidades pequeñas, por lo que la estimación de profundidad pierde precisión. Una baseline grande mejora la sensibilidad de la profundidad frente a pequeños cambios de disparidad, pero reduce el campo común y hace que las imágenes sean más diferentes, aumentando las oclusiones y las correspondencias ambiguas.

2.5. GEOMETRÍA EPIPOLAR: RESTRICCIÓN FUNDAMENTAL DEL PAR ESTÉREO

La geometría epipolar describe las relaciones geométricas entre dos vistas de una misma escena. Dado un punto observado en la imagen izquierda, su correspondencia en la imagen derecha no puede aparecer en cualquier lugar,

debe encontrarse sobre una línea concreta llamada línea epipolar. Esta restricción reduce el espacio de búsqueda y permite descartar correspondencias imposibles desde el punto de vista geométrico [1].

El plano que contiene el punto 3D y los centros ópticos de ambas cámaras se denomina plano epipolar. La intersección de ese plano con cada plano de imagen genera las líneas epipolares. Los epipolos son los puntos donde la línea base entre cámaras corta los planos de imagen, o equivalentemente la proyección del centro óptico de una cámara sobre la imagen de la otra.

La restricción epipolar se expresa mediante la matriz fundamental F . Si m_L y m_R son puntos correspondientes en coordenadas homogéneas en la imagen izquierda y derecha, debe cumplirse:

$$m_R^T \cdot F \cdot m_L = 0$$

La matriz fundamental resume la geometría epipolar entre dos imágenes sin requerir necesariamente conocer todos los parámetros de calibración. Cuando las cámaras están calibradas, se trabaja habitualmente con coordenadas normalizadas y aparece la matriz esencial E , relacionada con la rotación R y la traslación t entre cámaras:

$$E = [t]_x \cdot R$$

La matriz esencial permite expresar la misma restricción en coordenadas normalizadas. Su utilidad es que separa la geometría de la cámara de la geometría del movimiento relativo entre cámaras. En un sistema calibrado, F y E están relacionadas por las matrices intrínsecas de ambas cámaras:

$$E = K_R^T \cdot F \cdot K_L$$

En aplicaciones reales, F o E pueden estimarse a partir de correspondencias iniciales entre puntos. Como estas correspondencias pueden contener errores,

se emplean métodos robustos como RANSAC, que selecciona subconjuntos aleatorios, estima el modelo geométrico y conserva las correspondencias que cumplen la restricción con un error bajo.

2.6. RECTIFICACIÓN EPIPOLAR

La rectificación epipolar es una transformación aplicada a las imágenes del par estéreo para que las líneas epipolares queden alineadas horizontalmente. Tras la rectificación, un punto de la imagen izquierda debe buscarse en la imagen derecha sobre la misma fila [2]. Esto simplifica enormemente el problema de correspondencia, porque ya no se busca en una región 2D de la imagen, sino a lo largo de una única línea horizontal.

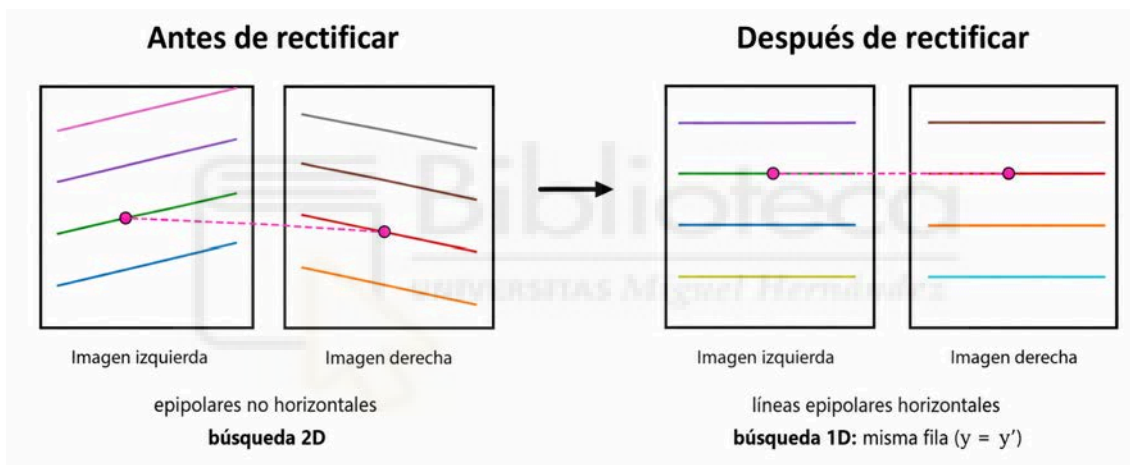


Figura 2.4. Efecto de la rectificación epipolar sobre las líneas de búsqueda.

El proceso de rectificación se basa en la calibración del sistema o en la geometría epipolar estimada. A grandes rasgos, se calculan dos transformaciones de remapeo, una para cada imagen, de forma que ambas queden proyectadas sobre planos de imagen virtuales coplanares y alineados.

En ese nuevo sistema, las correspondencias tienen la misma coordenada vertical:

$$vL = vR$$

La rectificación no reconstruye la escena por sí misma, pero facilita la etapa crítica de correspondencia. En datasets como KITTI, los pares estéreo se proporcionan ya rectificados, por lo que muchos métodos, tanto clásicos como basados en redes neuronales, asumen directamente que la disparidad es un desplazamiento horizontal.

Aspecto	Ventaja de rectificar	Posible inconveniente
Búsqueda de correspondencias	Reduce la búsqueda a una dimensión, lo que acelera y simplifica el algoritmo.	Si la calibración es mala, el remapeo introduce errores y la correspondencia empeora.
Cálculo de disparidad	Permite definir la disparidad como diferencia horizontal entre píxeles.	Puede recortar o deformar zonas de la imagen para alinear los planos virtuales.
Implementación	Facilita métodos densos como Block Matching, SGBM y CNN estéreo.	Requiere disponer de calibración o estimar correctamente la geometría epipolar.

2.7. CORRESPONDENCIA ESTÉREO

La correspondencia estéreo consiste en determinar qué punto, píxel o región de la imagen derecha corresponde al mismo punto físico observado en la imagen izquierda. Es el paso más delicado del método clásico, ya que la reconstrucción 3D posterior depende directamente de la calidad de estas correspondencias.

En imágenes rectificadas, la correspondencia se formula como una búsqueda horizontal. Si un punto aparece en la posición (u_L, v) en la imagen izquierda, se buscan candidatos en posiciones (u_R, v) de la imagen derecha. La disparidad se define como la diferencia horizontal entre ambas posiciones. En muchas aplicaciones se toma:

$$d = u_L - u_R$$

Con esta convención, los objetos cercanos suelen tener disparidades mayores y los objetos lejanos disparidades menores. En otros textos puede aparecer el signo contrario, pero lo importante es mantener una convención coherente.

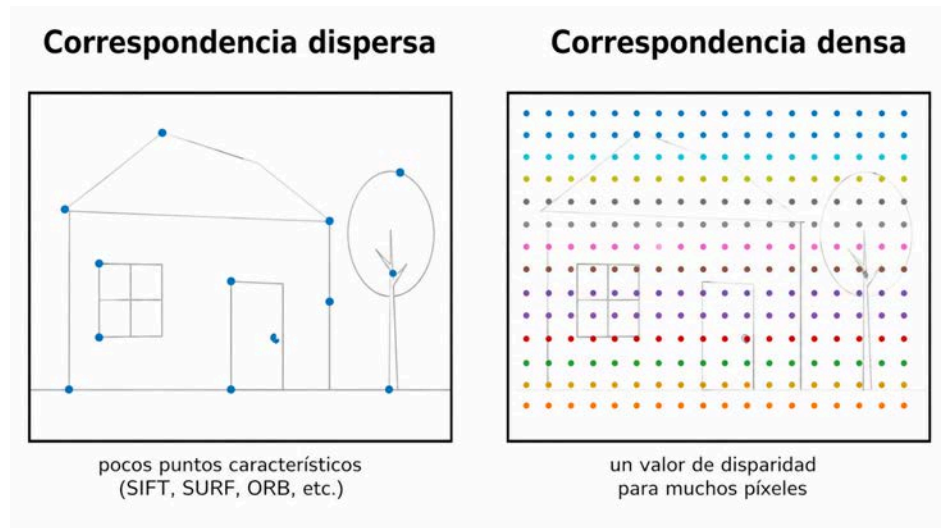


Figura 2.5. Diferencia entre correspondencia dispersa basada en características y correspondencia densa.

2.7.1. CORRESPONDENCIA DISPERSA BASADA EN CARACTERÍSTICAS

La correspondencia dispersa se basa en detectar puntos característicos en ambas imágenes y emparejarlos mediante descriptores. No intenta asignar profundidad a todos los píxeles, sino únicamente a puntos suficientemente informativos, como esquinas, bordes o zonas con alto gradiente.

El proceso habitual es:

1. Detectar puntos de interés en cada imagen. Pueden ser esquinas, regiones con textura o puntos de alto gradiente.
2. Calcular un descriptor para cada punto. El descriptor resume la apariencia local alrededor del punto.
3. Comparar los descriptores de la imagen izquierda y derecha mediante una distancia.
4. Filtrar correspondencias incorrectas usando criterios como el ratio test, la restricción epipolar o RANSAC.
5. Triangular las correspondencias aceptadas para obtener una nube de puntos dispersa.

SIFT y SURF son ejemplos clásicos de descriptores locales. SIFT se basa en características invariantes a escala y rotación, y describe la distribución local de gradientes. SURF persigue una filosofía similar, pero busca reducir el coste computacional mediante aproximaciones más rápidas, empleando respuestas tipo Haar y estructuras de escala eficientes. Otros descriptores, como ORB o BRIEF, utilizan descripciones binarias y se comparan con distancia de Hamming, lo que los hace más rápidos y adecuados para sistemas con recursos limitados.

Descrip tor	Tipo	Ventajas	Desventajas
SIFT	Descriptor métrico basado en gradientes	Robusto ante cambios de escala, rotación y cierta variación de iluminación. Adecuado para correspondencia dispersa robusta.	Mayor coste computacional que descriptores binarios. Produce una nube dispersa, no un mapa denso.
SURF	Descriptor métrico basado en respuestas tipo Haar	Más rápido que SIFT en muchas implementaciones y robusto ante cambios moderados.	Puede perder detalle frente a SIFT y también genera correspondencias dispersas.
ORB / BRIEF	Descriptores binarios	Muy rápidos y comparables con distancia Hamming. Útiles en tiempo real.	Menos robustos en cambios fuertes de escala, punto de vista o iluminación.

La correspondencia de características puede realizarse por fuerza bruta, comparando cada descriptor con todos los descriptores de la otra imagen, o mediante métodos aproximados como FLANN, que aceleran la búsqueda en espacios de alta dimensión. Para descriptores métricos se usan distancias como la euclídea o la del coseno; para descriptores binarios se usa habitualmente la distancia de Hamming.

La principal ventaja de los métodos basados en características es su robustez relativa: se concentran en zonas con mucha información y no dependen tanto de comparar ventanas completas de intensidad. Su principal inconveniente es que la reconstrucción resultante suele ser dispersa. Esto puede ser suficiente para estimar la pose, calibrar o reconstruir estructuras principales, pero no proporciona directamente un mapa de profundidad denso de toda la escena.

2.7.2. CORRESPONDENCIA DENSA BASADA EN ÁREA

Los métodos densos buscan estimar una disparidad para la mayor cantidad posible de píxeles. En lugar de comparar únicamente puntos característicos, comparan ventanas o regiones de la imagen izquierda con posibles ventanas desplazadas en la imagen derecha. La idea básica consiste en evaluar una función de coste para cada píxel p y cada disparidad candidata d .

Una función de coste clásica es la suma de diferencias cuadráticas en una ventana W :

$$SSD(p, d) = \sum_{q \in W(p)} [I_L(q) - I_R(q - d)]^2$$

También pueden usarse otras funciones, como SAD, NCC o variantes normalizadas que reducen la sensibilidad a cambios de iluminación. Para cada píxel se elige la disparidad que minimiza el coste o maximiza la similitud.

Posteriormente se aplican filtros y regularización para eliminar correspondencias aisladas o incoherentes.

Familia	Descripción	Ventajas	Limitaciones
Block Matching	Compara ventanas locales para distintas disparidades y elige el mejor desplazamiento.	Simple, rápido y fácil de paralelizar. Puede producir mapas densos.	Sensible a ruido, iluminación, baja textura, discontinuidades y tamaño de ventana.
Semi-Global Matching (SGM / SGBM)	Agrega costes a lo largo de múltiples direcciones y penaliza cambios bruscos de disparidad.	Mejor equilibrio entre precisión y coste computacional que métodos puramente locales.	Sigue fallando en oclusiones, reflejos, baja textura y zonas repetitivas. Requiere ajustar parámetros.
Métodos globales	Formulan la disparidad como optimización energética global.	Pueden generar mapas más coherentes.	Suelen tener mayor coste computacional.

2.8. MAPA DE DISPARIDAD Y REGULARIZACIÓN

El resultado intermedio más importante en visión estéreo es el mapa de disparidad. Es una imagen en la que cada píxel almacena el desplazamiento horizontal necesario para encontrar su correspondencia en la otra cámara. Si el mapa de disparidad es denso, puede transformarse en un mapa de profundidad y posteriormente en una nube de puntos.

El cálculo píxel a píxel mediante comparación local suele producir ruido. Por ello, los métodos clásicos incorporan regularización. Semi-Global Matching es una técnica representativa: combina un coste local de coincidencia con penalizaciones que favorecen mapas de disparidad suaves, pero permite discontinuidades en bordes de objetos [3]. De forma simplificada, la energía a minimizar puede expresarse como:

$$E(D) = \sum_p C(p, D_p) + \sum_{(p,q)} P_1 \cdot I[|D_p - D_q| = 1] + \sum_{(p,q)} P_2 \cdot I[|D_p - D_q| > 1]$$

En esta expresión, $C(p, D_p)$ es el coste de asignar la disparidad D_p al píxel p . El término P_1 penaliza cambios pequeños de disparidad entre píxeles vecinos, mientras que P_2 penaliza cambios grandes. Esta formulación favorece superficies suaves, pero no obliga a que toda la imagen tenga la misma profundidad. Por eso SGM/SGBM suele comportarse mejor que un Block Matching puramente local en escenas reales.

Después de calcular la disparidad, es habitual aplicar postprocesado: comprobación izquierda-derecha, eliminación de regiones espurias, filtrado de speckles, interpolación de huecos y suavizado conservando bordes. Estos pasos no resuelven completamente las ambigüedades, pero mejoran la coherencia visual del mapa de disparidad.

2.9. TRIANGULACIÓN, PROFUNDIDAD Y NUBE DE PUNTOS

Una vez conocida la correspondencia entre dos proyecciones de un mismo punto, se puede recuperar su posición 3D mediante triangulación. La

triangulación consiste en calcular el punto del espacio donde se cruzan, o se aproximan, los rayos que salen desde los centros ópticos de ambas cámaras y pasan por los puntos de imagen correspondientes [1].

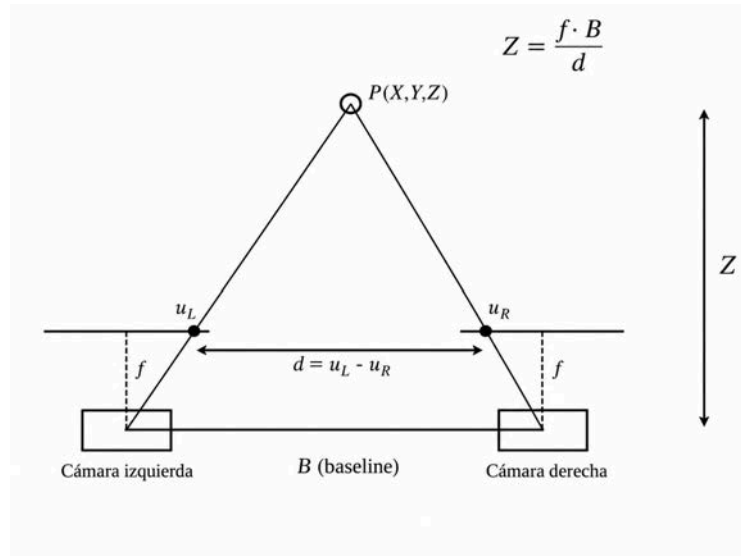


Figura 2.6. Relación geométrica entre baseline, disparidad y profundidad en un par estéreo rectificado.

En un par estéreo rectificado con ejes alineados, la reconstrucción se simplifica. Si se conoce la distancia focal f y la baseline B , la profundidad Z se obtiene a partir de la disparidad d :

$$Z = \frac{f \cdot B}{d}$$

Esta relación muestra que la profundidad es inversamente proporcional a la disparidad. Un objeto cercano produce una gran diferencia de posición entre ambas imágenes, mientras que un objeto lejano produce una disparidad pequeña. Si además se conocen los parámetros intrínsecos, cada píxel con disparidad válida puede proyectarse al espacio 3D mediante:

$$X = \frac{(u - c_x) \cdot Z}{f_x} \quad Y = \frac{(v - c_y) \cdot Z}{f_y} \quad Z = \frac{f_x \cdot B}{d}$$

Cuando las cámaras no están alineadas o se trabaja con ejes convergentes sin reducir el problema a la forma rectificada, la triangulación se formula de manera más general mediante las matrices de proyección P y P' . Para cada correspondencia, se plantea un sistema lineal que busca el punto 3D M que mejor satisface las proyecciones observadas en ambas cámaras:

$$m_L \approx P \cdot M \quad y \quad m_R \approx P' \cdot M$$

Debido al ruido de medición, los rayos de proyección no suelen cruzarse exactamente en un mismo punto. Por ello, se calcula una solución aproximada, por ejemplo mediante mínimos cuadrados o minimizando el error de reproyección. El error de reproyección mide la distancia entre los puntos observados en las imágenes y los puntos que resultan de volver a proyectar la reconstrucción 3D estimada.

Aplicando estas ecuaciones a todos los píxeles con disparidad válida se obtiene una nube de puntos. Cada punto de la nube almacena una posición espacial (X, Y, Z), y opcionalmente puede incorporar color tomando el valor RGB del píxel correspondiente de la imagen izquierda. La nube de puntos permite visualizar la geometría de la escena y constituye una representación útil para análisis, navegación, inspección o reconstrucción del entorno.

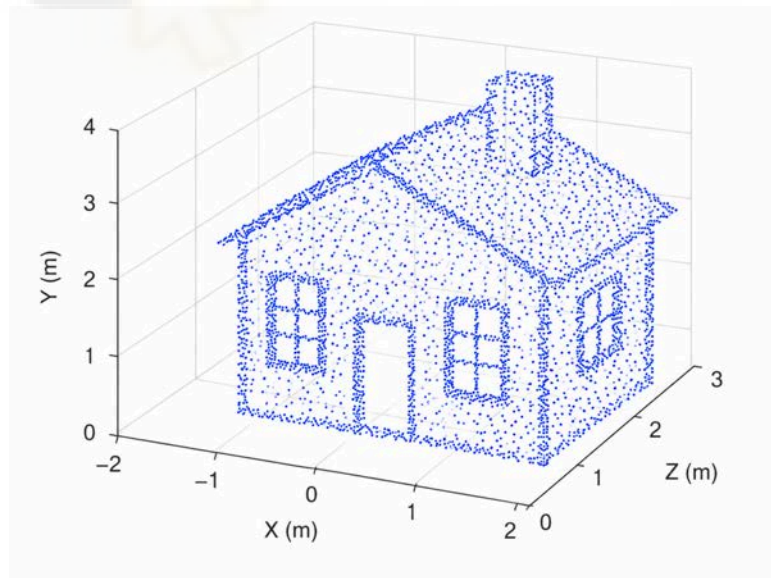


Figura 2.7. Representación conceptual de una nube de puntos obtenida a partir de un mapa de disparidad.

2.10. LIMITACIONES DEL MÉTODO CLÁSICO

Aunque el método clásico o geométrico tiene una base geométrica clara y no requiere entrenamiento, presenta limitaciones importantes en escenas reales. La correspondencia puede ser ambigua o imposible en varias situaciones:

- Zonas con baja textura, como paredes lisas, cielo o carreteras uniformes. En estas regiones muchas ventanas se parecen entre sí y el coste de correspondencia no permite decidir con claridad.
- Patrones repetitivos, como vallas, ventanas o marcas similares. El algoritmo puede emparejar el punto con una repetición incorrecta del patrón.
- Oclusiones, donde un punto visible en una cámara no aparece en la otra. En ese caso no existe una correspondencia real que pueda encontrarse.
- Superficies reflectantes o transparentes, porque la apariencia del punto puede cambiar entre cámaras.
- Cambios de iluminación o exposición entre las dos imágenes, que afectan a métodos basados en intensidad.
- Bordes de objetos y estructuras finas, donde pequeños errores de correspondencia provocan saltos de profundidad.

También existe una limitación geométrica: la profundidad se calcula como $Z = fB/d$. Cuando d es pequeño, un error de pocos píxeles puede provocar un gran error en Z . Por ello, los puntos lejanos suelen ser más difíciles de reconstruir con precisión. Aumentar la baseline mejora la precisión de profundidad, pero también hace que las imágenes sean más diferentes y aumenta las oclusiones. Este compromiso es una de las razones por las que el diseño físico del sistema estéreo es importante.

2.11. APRENDIZAJE PROFUNDO APLICADO A VISIÓN ESTÉREO

Las redes neuronales convolucionales han cambiado la forma de abordar la estimación de disparidad. En los métodos clásicos, el diseñador define manualmente los descriptores, la función de coste, la agregación y la regularización. En cambio, los métodos basados en aprendizaje profundo aprenden parte de estas operaciones a partir de datos.

Los primeros enfoques con redes neuronales convolucionales, también conocidas como CNN de sus siglas en inglés, se centraron en aprender una medida de similitud entre parches de la imagen izquierda y derecha [5]. Posteriormente, las arquitecturas de extremo a extremo comenzaron a recibir directamente el par estéreo completo y a producir un mapa de disparidad. La estructura habitual de estas redes incluye extracción de características, construcción de un volumen de coste y regularización de la disparidad.

El volumen de coste es una representación que almacena, para cada posición de la imagen, distintas hipótesis de disparidad. En lugar de elegir directamente una correspondencia local, la red procesa simultáneamente muchas hipótesis y aprende a seleccionar la más coherente teniendo en cuenta información local y contextual. Esto permite mejorar el rendimiento en zonas ambiguas, aunque no elimina completamente problemas como oclusiones, baja textura o cambios de dominio.

Arquitecturas como GC-Net introdujeron el uso de volúmenes de coste y convoluciones 3D para estimación estéreo de extremo a extremo [6]. PSMNet añadió módulos de extracción multiescala, como Spatial Pyramid Pooling, y estructuras hourglass 3D para regularizar el volumen de coste [7]. Este tipo de modelos puede superar a muchos métodos clásicos en precisión, pero requiere datos de entrenamiento, mayor memoria y capacidad de cálculo.

Aspecto	Métodos clásicos	Métodos con CNN
Conocimiento utilizado	Geometría explícita, funciones de coste y reglas diseñadas manualmente.	Representaciones aprendidas a partir de datos.
Entrenamiento	No necesitan entrenamiento previo.	Requieren datasets y ajuste de pesos.
Interpretabilidad	Mayor claridad geométrica y control de parámetros.	Menor interpretabilidad interna, aunque mantienen relación con disparidad y coste.
Rendimiento en escenas difíciles	Limitado en baja textura, reflejos, repetición y oclusiones.	Puede mejorar usando el contexto aprendido, pero sigue teniendo limitaciones.
Coste computacional	Normalmente menor, especialmente en implementaciones optimizadas.	Mayor coste, sobre todo con volúmenes de coste 3D.

2.12. DATASETS, BENCHMARKS Y MÉTRICAS DE EVALUACIÓN

Para evaluar métodos de visión estéreo se utilizan datasets con pares de imágenes y mapas de disparidad de referencia. Los datasets sintéticos permiten generar grandes cantidades de datos con disparidades densas, mientras que los datasets reales proporcionan escenas capturadas en condiciones más cercanas a la aplicación final.

KITTI es uno de los bancos de pruebas más utilizados en visión estéreo aplicada a escenas exteriores. Aunque está orientado al entorno de conducción, resulta útil para este trabajo porque proporciona pares estéreo rectificados, información de calibración y mapas de disparidad de referencia en sus conjuntos de entrenamiento. Esto permite comparar de forma objetiva tanto métodos clásicos como redes neuronales.

Las métricas habituales comparan la disparidad predicha con la disparidad real. El End Point Error mide el error absoluto medio en píxeles. D1-all contabiliza el porcentaje de píxeles cuyo error supera un umbral absoluto y relativo. También pueden utilizarse métricas relativas como AbsRel o umbrales de precisión como delta1.

2.13. SÍNTESIS DEL ESTADO DEL ARTE Y RELACIÓN CON EL TRABAJO

En resumen, la reconstrucción 3D estéreo clásica se apoya en una cadena geométrica formada por calibración, corrección de distorsión, rectificación epipolar, correspondencia, cálculo de disparidad y triangulación. Su principal fortaleza es que cada etapa tiene una interpretación matemática clara. Esto permite entender cómo influyen la focal, la baseline, la disposición de las cámaras y la calidad de las correspondencias en la reconstrucción final.

Sin embargo, las técnicas clásicas dependen de la calidad de la correspondencia. En escenas reales, la presencia de oclusiones, regiones sin textura, patrones repetitivos o cambios fotométricos puede generar errores importantes. Los métodos basados en aprendizaje profundo surgen como una alternativa capaz de aprender representaciones más robustas y regularizar la disparidad a partir de datos, aunque a costa de necesitar entrenamiento y más recursos computacionales.

Por este motivo, en el contexto de este trabajo resulta adecuado estudiar ambos enfoques: por un lado, un método clásico como referencia geométrica y comparativa; por otro, una CNN estéreo como PSMNet, capaz de estimar mapas de disparidad densos a partir de pares de imágenes rectificadas. La comparación entre ambos permite analizar no solo la precisión final, sino también las ventajas y limitaciones de cada familia de métodos en la reconstrucción 3D estereoscópica.



3. MATERIALES Y MÉTODOS

3.1. ENTORNO DE DESARROLLO

El desarrollo experimental de este trabajo se ha realizado utilizando Google Colab como entorno principal de ejecución. Esta plataforma permite trabajar en la nube y acceder a recursos de computación avanzados sin depender directamente de las prestaciones del ordenador local desde el que se desarrolla el proyecto.

Durante los experimentos se ha empleado una GPU NVIDIA A100 Tensor Core, lo que ha permitido reducir considerablemente los tiempos de entrenamiento y evaluación de la red neuronal. Este aspecto resulta especialmente importante en tareas de aprendizaje profundo, donde el procesamiento de grandes volúmenes de imágenes y el entrenamiento de redes convolucionales requieren una elevada capacidad de cálculo.

3.2. LIBRERÍAS Y HERRAMIENTAS UTILIZADAS

El proyecto se ha desarrollado en Python, utilizando distintas librerías y herramientas para cubrir todas las fases del trabajo: preparación de datos, entrenamiento de la CNN, evaluación cuantitativa, implementación del método clásico, comparación visual y reconstrucción tridimensional. A continuación, se describen las principales librerías, módulos y herramientas utilizadas en los códigos desarrollados.

- Python: lenguaje de programación utilizado para implementar todos los scripts del proyecto. Se ha empleado por su amplia disponibilidad de librerías orientadas a aprendizaje profundo, visión por computador, cálculo numérico, visualización y gestión de archivos.
- Google Colab: entorno de ejecución utilizado para desarrollar y ejecutar los códigos del proyecto. Ha permitido trabajar con GPU, instalar librerías adicionales, acceder a Google Drive y ejecutar los procesos de

entrenamiento, evaluación y reconstrucción 3D sin depender de hardware local específico.

- Google Drive: sistema de almacenamiento empleado para guardar datasets, pesos preentrenados, checkpoints generados durante el fine-tuning, métricas, configuraciones, mapas de disparidad, figuras comparativas y resultados de reconstrucción 3D.
- google.colab.drive: módulo utilizado para montar Google Drive dentro del entorno de Colab. Gracias a ello, los códigos pueden acceder a las carpetas del proyecto, leer los datasets y guardar los resultados de forma persistente.
- google.colab.runtime: módulo utilizado en algunos scripts para liberar el entorno de ejecución al finalizar procesos largos, evitando mantener ocupada la GPU de Colab innecesariamente.
- PyTorch: librería principal utilizada para trabajar con la red neuronal convolucional. Se ha empleado para crear el modelo PSMNet, cargar pesos preentrenados, ejecutar inferencia, realizar fine-tuning, calcular métricas y gestionar tensores en CPU o GPU.
- torch: módulo base de PyTorch utilizado para la creación y manipulación de tensores, envío de datos a GPU, comprobación de valores finitos, cálculo de errores y ejecución de operaciones numéricas durante entrenamiento, evaluación e inferencia.
- torch.nn: módulo de PyTorch utilizado para gestionar la estructura del modelo neuronal. En el proyecto se usa especialmente para tratar el modelo como una arquitectura entrenable y para controlar qué parámetros quedan congelados o activos durante los distintos experimentos de fine-tuning.

- `torch.nn.functional`: módulo utilizado para funciones concretas de entrenamiento e inferencia. Se emplea para calcular la función de pérdida Smooth L1 Loss durante el entrenamiento y para aplicar padding a las imágenes antes de introducirlas en PSMNet cuando es necesario adaptar su tamaño.
- `torch.optim`: módulo utilizado para definir el optimizador del entrenamiento. En este proyecto se utiliza Adam, encargado de actualizar los pesos de las capas seleccionadas durante el fine-tuning.
- `torch.utils.data.DataLoader`: herramienta utilizada para cargar los pares estéreo y sus mapas de disparidad real por lotes. Permite definir el tamaño de batch, mezclar las muestras durante el entrenamiento, mantener el orden durante la evaluación y acelerar la carga de datos mediante varios procesos.
- `torch.no_grad`: contexto utilizado durante evaluación, inferencia y reconstrucción 3D para desactivar el cálculo de gradientes. Esto reduce el consumo de memoria y acelera la ejecución cuando no se están actualizando los pesos del modelo.
- `torch.cuda`: módulo utilizado para trabajar con GPU. Se emplea para comprobar si existe una GPU activa, mover el modelo y los tensores al dispositivo CUDA, sincronizar la GPU al medir tiempos de inferencia y liberar memoria cuando finaliza una evaluación.
- `torch.load`: función utilizada para cargar checkpoints desde disco. Permite recuperar pesos preentrenados o pesos obtenidos durante el fine-tuning.

- `torch.save`: función utilizada para guardar checkpoints durante el entrenamiento. Se emplea para almacenar los pesos cuando se alcanza una mejora en la métrica D1-all y también para guardar los pesos correspondientes a la última época ejecutada.
- `state_dict`: estructura de PyTorch que contiene los parámetros del modelo. En el proyecto se utiliza para guardar y recuperar los pesos de PSMNet.
- `load_state_dict`: función utilizada para insertar los pesos cargados dentro de la arquitectura de PSMNet. En los códigos se realiza una carga estricta, comprobando que no falten claves, que no existan claves inesperadas y que las dimensiones de los tensores coincidan con la arquitectura utilizada.
- CUDA: tecnología utilizada para ejecutar el entrenamiento y la inferencia de la CNN en GPU. Su uso es especialmente importante en PSMNet, ya que el entrenamiento y la estimación de disparidad son procesos computacionalmente costosos.
- `nvidia-smi`: herramienta utilizada mediante `subprocess` para comprobar el estado de la GPU disponible en Colab. Permite visualizar información de la tarjeta gráfica antes de iniciar procesos pesados.
- NumPy: librería utilizada para operaciones numéricas generales. Se emplea para trabajar con matrices, máscaras de validez, medias de métricas, conversión de datos, cálculo de profundidad, generación de coordenadas 3D, normalización de mapas y guardado o lectura de archivos en formato `.npy`.

- OpenCV: librería de visión por computador utilizada principalmente para el método clásico de correspondencia estéreo y para el tratamiento de imágenes. Se emplea para leer imágenes, leer mapas de disparidad KITTI, convertir espacios de color, guardar imágenes, aplicar mapas de color y generar visualizaciones de disparidad y error.
- cv2.StereoSGBM_create: herramienta de OpenCV utilizada para implementar el método clásico de correspondencia estéreo densa. Este método calcula un mapa de disparidad a partir del par estéreo sin utilizar una red neuronal.
- matcher.compute: función del método StereoSGBM que realiza el cálculo principal de disparidad en el método clásico. En la evaluación temporal se mide esta operación para compararla con la llamada de inferencia de PSMNet.
- cv2.imread: función utilizada para leer imágenes RGB, imágenes en escala de grises y mapas de disparidad en formato KITTI.
- cv2.imwrite: función utilizada para guardar mapas de disparidad, mapas de error, mapas de profundidad y otras imágenes generadas durante la evaluación y reconstrucción.
- cv2.applyColorMap: función utilizada para convertir mapas de disparidad, error o profundidad en imágenes coloreadas que faciliten la interpretación visual de los resultados.
- cv2.cvtColor: función utilizada para convertir imágenes entre formatos de color, especialmente de BGR a RGB cuando se leen imágenes con OpenCV y posteriormente se representan con Matplotlib.

- `cv2.circle`: función utilizada en el renderizado estático de nubes de puntos, dibujando cada punto proyectado sobre una imagen 2D desde una cámara virtual.
- `PIL.Image`: herramienta utilizada para abrir, convertir y guardar imágenes. Se emplea especialmente en la inferencia con PSMNet, ya que el preprocesado del repositorio oficial trabaja con imágenes PIL. También se utiliza para copiar imágenes de entrada y guardar vistas renderizadas.
- `Matplotlib`: librería utilizada para generar visualizaciones y figuras. En el proyecto se emplea para mostrar mapas de disparidad con barras de color, crear figuras comparativas entre método clásico y PSMNet, representar errores y guardar resultados visuales en formato PNG.
- `matplotlib.pyplot`: módulo de Matplotlib utilizado para crear figuras, ejes, mapas de color, barras de color y guardar imágenes finales mediante `savefig`.
- `copy`: módulo utilizado para copiar mapas de color de Matplotlib antes de modificarlos. Se emplea, por ejemplo, para definir un color específico en zonas inválidas sin alterar el mapa de color original.
- `Open3D`: librería utilizada para generar y guardar nubes de puntos 3D. En el proyecto se emplea para construir nubes de puntos coloreadas a partir de la disparidad estimada y guardarlas en formato `.ply`.
- `open3d.geometry.PointCloud`: estructura utilizada para crear la nube de puntos tridimensional.

- `open3d.utility.Vector3dVector`: herramienta utilizada para convertir las matrices de puntos y colores en el formato requerido por Open3D.
- `open3d.io.write_point_cloud`: función utilizada para guardar la nube de puntos en formato `.ply`, permitiendo su visualización posterior en software externo compatible.
- `Plotly`: librería utilizada para generar visualizaciones 3D interactivas. En el proyecto se emplea para crear nubes de puntos navegables en HTML.
- `plotly.graph_objects`: módulo utilizado para crear figuras interactivas de `Plotly`. En concreto, se utiliza `Scatter3d` para representar la nube de puntos mediante marcadores 3D.
- `go.Figure`: estructura utilizada para construir la visualización interactiva de la nube de puntos.
- `go.Scatter3d`: herramienta utilizada para representar los puntos 3D con sus coordenadas X, Y, Z y colores RGB.
- `write_html`: función utilizada para guardar la nube de puntos interactiva como archivo HTML, permitiendo abrirla posteriormente en un navegador.
- `os`: módulo utilizado para operaciones generales del sistema de archivos. Se emplea para comprobar rutas, crear rutas completas, cambiar directorios, comprobar tamaños de archivo y gestionar variables del entorno.

- `sys`: módulo utilizado para modificar el path de Python. Esto permite importar correctamente los módulos del repositorio de PSMNet desde Google Drive o desde una carpeta clonada en Colab.
- `pathlib.Path`: herramienta utilizada para gestionar rutas de archivos y carpetas de forma más clara y robusta. Se emplea para comprobar existencia de archivos, crear carpetas, recorrer directorios y obtener nombres de archivos.
- `shutil`: módulo utilizado para copiar archivos y eliminar carpetas. Se emplea, por ejemplo, para copiar datasets desde Google Drive al almacenamiento temporal de Colab y para limpiar directorios antes de descomprimir nuevos datos.
- `zipfile`: módulo utilizado para descomprimir los datasets KITTI almacenados en formato ZIP. También se emplea para comprobar si un archivo comprimido está corrupto antes de extraerlo.
- `csv`: módulo utilizado para guardar resultados numéricos en formato CSV. Se emplea para almacenar métricas por época, métricas por imagen, resúmenes de evaluación, métricas añadidas y métricas de tiempo.
- `json`: módulo utilizado para guardar configuraciones y resúmenes de ejecución. Permite registrar parámetros de entrenamiento, capas entrenables, rutas de pesos, configuración de evaluación, información de calibración y archivos generados.
- `time`: módulo utilizado para medir tiempos de ejecución. Se emplea para calcular la duración de épocas de entrenamiento, el tiempo de inferencia de PSMNet, el tiempo de cálculo del método clásico y el tiempo total de reconstrucción.

- random: módulo utilizado para fijar semillas aleatorias en el entrenamiento. Junto con NumPy y PyTorch, permite mejorar la reproducibilidad de los experimentos.
- subprocess: módulo utilizado para ejecutar comandos del sistema, principalmente nvidia-smi para comprobar la GPU disponible.
- gdown: herramienta utilizada para descargar el checkpoint preentrenado de PSMNet desde Google Drive cuando no está disponible en la ruta local prevista.
- Git y GitHub: herramientas utilizadas para obtener el repositorio oficial de PSMNet cuando no se encuentra disponible en Drive. El repositorio se usa como base para importar la arquitectura de la red y sus funciones auxiliares.
- Repositorio oficial de PSMNet: base principal de la implementación de la CNN utilizada. De este repositorio se importan la arquitectura stackhourglass, el cargador myImageFloder y el módulo preprocess.
- stackhourglass: arquitectura concreta de PSMNet utilizada en el proyecto. Se instancia con un valor máximo de disparidad y constituye el modelo encargado de estimar el mapa de disparidad a partir de un par de imágenes estéreo.
- myImageFloder: cargador de datos procedente del repositorio de PSMNet. Se utiliza durante entrenamiento y evaluación para leer imágenes izquierda y derecha junto con sus mapas de disparidad real.

- preprocess: módulo del repositorio de PSMNet utilizado para aplicar las transformaciones necesarias a las imágenes antes de introducirlas en la red durante la inferencia y la reconstrucción 3D.
- Checkpoints: archivos donde se almacenan los pesos del modelo y, en algunos casos, el estado del optimizador y las métricas asociadas. Se utilizan para cargar el modelo preentrenado, guardar los pesos obtenidos durante el fine-tuning y evaluar posteriormente distintas versiones del modelo.
- Archivos .tar: formato utilizado para guardar los checkpoints de PyTorch. En el proyecto se emplean para almacenar los pesos preentrenados, los pesos de la mejor época según D1-all, los pesos de la última época y los pesos seleccionados para la reconstrucción 3D.
- Archivos .csv: formato utilizado para almacenar métricas y resultados tabulados. Facilitan el análisis posterior y la generación de tablas o gráficas comparativas.
- Archivos .json: formato utilizado para guardar configuraciones, resúmenes de experimentos y metadatos de reconstrucción. Permiten documentar las condiciones exactas bajo las que se ha ejecutado cada proceso.
- Archivos .npy: formato utilizado para guardar matrices NumPy. Se emplea para almacenar mapas de disparidad predicha, mapas de profundidad y predicciones del método clásico conservando los valores numéricos originales.
- Archivos .png: formato utilizado para guardar imágenes de entrada, mapas de disparidad, mapas de profundidad, mapas de error y figuras comparativas.

- Archivos .ply: formato utilizado para guardar nubes de puntos 3D coloreadas generadas a partir de la disparidad estimada.
- Archivos .html: formato utilizado para guardar visualizaciones interactivas de nubes de puntos 3D generadas con Plotly.

En conjunto, estas librerías y herramientas permiten implementar el flujo completo del proyecto: preparación de datasets, carga de PSMNet, fine-tuning de distintas configuraciones de capas, evaluación cuantitativa de la CNN, comparación con un método clásico, generación de figuras visuales, conversión de disparidad a profundidad y reconstrucción tridimensional mediante nubes de puntos.

3.3. ALMACENAMIENTO Y ORGANIZACIÓN DE ARCHIVOS

Los archivos de entrada y salida utilizados en Google Colab se han gestionado mediante Google Drive. De esta forma, los datasets, pesos preentrenados, modelos obtenidos durante el entrenamiento, métricas, mapas de disparidad, figuras comparativas y resultados de reconstrucción 3D se almacenan de forma persistente y organizada.

Esta organización permite conservar los resultados aunque finalice la sesión de Colab, así como reutilizar los archivos generados en diferentes fases del proyecto. Además, facilita la separación entre los datos de entrenamiento, los pesos del modelo y los resultados obtenidos en la evaluación.

La carpeta *PSMNet-master* contiene el repositorio de PSMNet empleado como base para importar la arquitectura de la red, el cargador de datos y las funciones de preprocesado necesarias para la inferencia.

La carpeta *Entrenamiento* almacena los pesos preentrenados y los resultados generados durante el fine-tuning. Dentro de ella se separan los tres experimentos realizados: entrenamiento de todas las capas, entrenamiento de capas intermedias y finales, y entrenamiento de solo capas finales. En cada experimento se guardan los pesos asociados a las épocas con mejora de D1-all y los pesos correspondientes a la última época ejecutada.

La carpeta *Evaluacion* almacena los resultados de la evaluación cuantitativa. Se divide en la evaluación del modelo preentrenado, la evaluación de los tres modelos ajustados mediante fine-tuning, la evaluación del método clásico y la comparación visual entre el método clásico y la CNN. En los experimentos de fine-tuning se separan los resultados obtenidos con los pesos de mejor D1-all y con los pesos de la última época.

La carpeta *MetodoClasico* contiene los resultados obtenidos mediante el método de correspondencia estéreo tradicional. Dentro de ella se almacenan tanto las métricas como los mapas de disparidad generados. Los mapas se organizan en subcarpetas según su finalidad: predicción numérica, predicción en formato KITTI, visualización de la predicción, visualización de la disparidad real y visualización del error.

La carpeta *ComparacionMetodoclasicoVsCnn* contiene las figuras utilizadas para comparar visualmente el método clásico con PSMNet.

La carpeta *Reconstrucción3d* contiene los pesos seleccionados para la reconstrucción tridimensional y los resultados finales generados. Dentro de *Resultados*, cada muestra procesada dispone de su propia carpeta, donde se almacenan las imágenes de entrada, el mapa de disparidad predicho, el mapa de profundidad, la nube de puntos, la visualización interactiva y las vistas estáticas generadas.

Además de estas carpetas de Google Drive, los códigos utilizan carpetas temporales dentro del entorno de Colab para copiar y descomprimir los datasets antes de procesarlos. Estas carpetas temporales no forman parte de la estructura persistente del proyecto, ya que se eliminan al finalizar o reiniciar la sesión de Colab.

3.4. MÉTODO BASADO EN PSMNET

PSMNet, o Pyramid Stereo Matching Network, es la red neuronal convolucional utilizada en este trabajo, en concreto la variante *stackhourglass*, correspondiente al modelo con estructuras hourglass apiladas en la etapa de regularización 3D [8]. Esta variante se diferencia de una versión más básica de PSMNet en que no utiliza una única red 3D simple para refinar el volumen de coste, sino varios módulos hourglass conectados de forma sucesiva. Esto permite procesar el volumen de coste a distintas resoluciones y refinar progresivamente la estimación de disparidad.

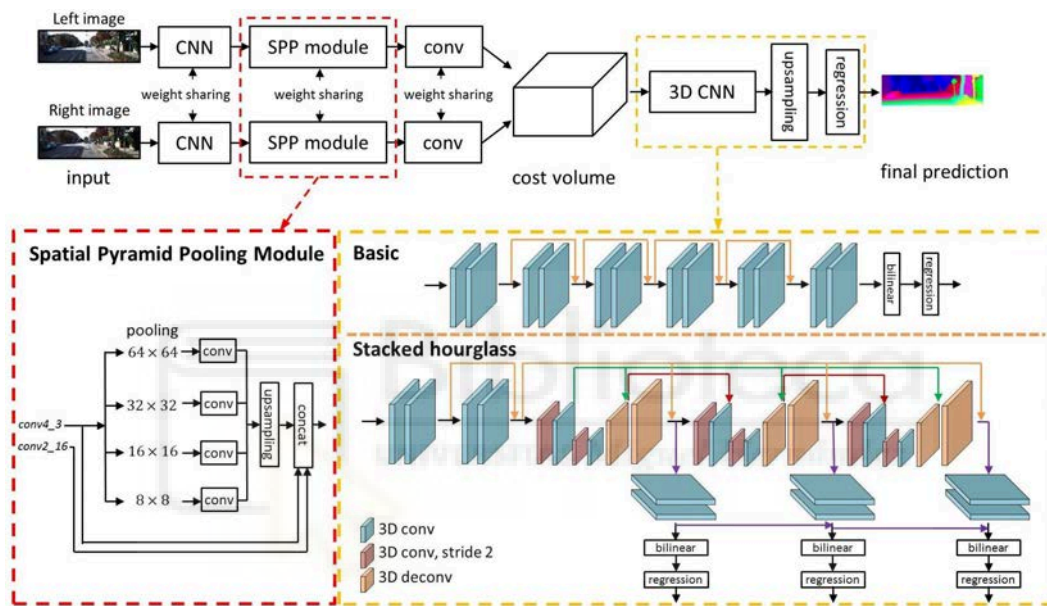


Figura 3.1. Esquema general de la arquitectura de PSMnet.

Uno de los elementos principales de PSMNet es el módulo Spatial Pyramid Pooling, que permite capturar información a diferentes escalas [7]. Esto ayuda a la red a considerar no solo detalles locales, sino también información más amplia de la escena, lo que resulta útil en regiones con baja textura o correspondencias ambiguas.

La arquitectura puede dividirse en varios grupos de capas. En primer lugar, el bloque *feature_extraction* se encarga de extraer características de la imagen izquierda y de la imagen derecha. Dentro de este bloque se encuentra *firstconv*, que realiza las primeras convoluciones sobre la imagen RGB de entrada y genera los primeros mapas de características.

A continuación, se encuentran las capas residuales *layer1*, *layer2*, *layer3* y *layer4*. Estas capas continúan el proceso de extracción de características. Las primeras capas, *layer1* y *layer2*, obtienen descriptores más próximos a la imagen original, mientras que *layer3* y *layer4* generan representaciones más profundas y con mayor contexto espacial.

Después se aplica el módulo Spatial Pyramid Pooling, formado por las ramas *branch1*, *branch2*, *branch3* y *branch4*. Estas ramas agregan información a distintas escalas, permitiendo que la red combine detalles locales con información más global de la escena. Esta etapa es especialmente útil en zonas con poca textura, superficies uniformes o regiones donde la correspondencia entre imágenes resulta ambigua.

La capa *lastconv* fusiona las características obtenidas por las ramas de la pirámide espacial junto con las características principales extraídas previamente. Su salida se utiliza para construir el volumen de coste, donde se comparan las características de la imagen izquierda y derecha para diferentes valores posibles de disparidad.

Una vez construido el volumen de coste, este se procesa mediante bloques de regularización 3D. En la implementación utilizada aparecen los bloques *dres0*, *dres1*, *dres2*, *dres3* y *dres4*. Estos bloques aplican convoluciones tridimensionales sobre el volumen de coste para mejorar la coherencia de las correspondencias y refinar la estimación de disparidad. En la variante *stackhourglass*, varios de estos bloques forman la estructura de hourglass apilados.

Finalmente, las capas *classif1*, *classif2* y *classif3* generan las predicciones de disparidad utilizadas durante el entrenamiento. La red produce varias salidas intermedias, lo que permite aplicar supervisión en distintas etapas. Durante la inferencia se utiliza la predicción final, correspondiente a la salida más refinada.

A partir de esta estructura se plantean tres experimentos de fine-tuning para analizar el efecto de ajustar diferentes partes de la red:

- Todas las capas: se actualizan todos los parámetros del modelo, incluyendo *firstconv*, *layer1*, *layer2*, *layer3*, *layer4*, *branch1-branch4*, *lastconv*, *dres0-dres4* y *classif1-classif3*.
- Capas intermedias y finales: se congelan las capas iniciales *firstconv*, *layer1* y *layer2*, y se actualizan *layer3*, *layer4*, *branch1-branch4*, *lastconv*, *dres0-dres4* y *classif1-classif3*.
- Solo capas finales: se congelan las capas de extracción inicial e intermedia, es decir, *firstconv*, *layer1*, *layer2*, *layer3* y *layer4*. En este caso se actualizan únicamente *branch1-branch4*, *lastconv*, *dres0-dres4* y *classif1-classif3*.



3.5. PESOS PREENTRENADOS Y TRANSFERENCIA DE APRENDIZAJE

En este trabajo no se entrena PSMNet desde cero. Se parte de pesos preentrenados en Scene Flow, un conjunto sintético de gran tamaño utilizado habitualmente en tareas de estimación de disparidad. Scene Flow agrupa varios subconjuntos, entre los que destacan FlyingThings3D, formado por objetos 3D en flotantes; Monkaa, basado en escenas sintéticas de animación; y Driving, compuesto por escenas de conducción generadas de forma sintética [9]. El uso de pesos preentrenados permite aprovechar el conocimiento previo adquirido por la red en una tarea similar.

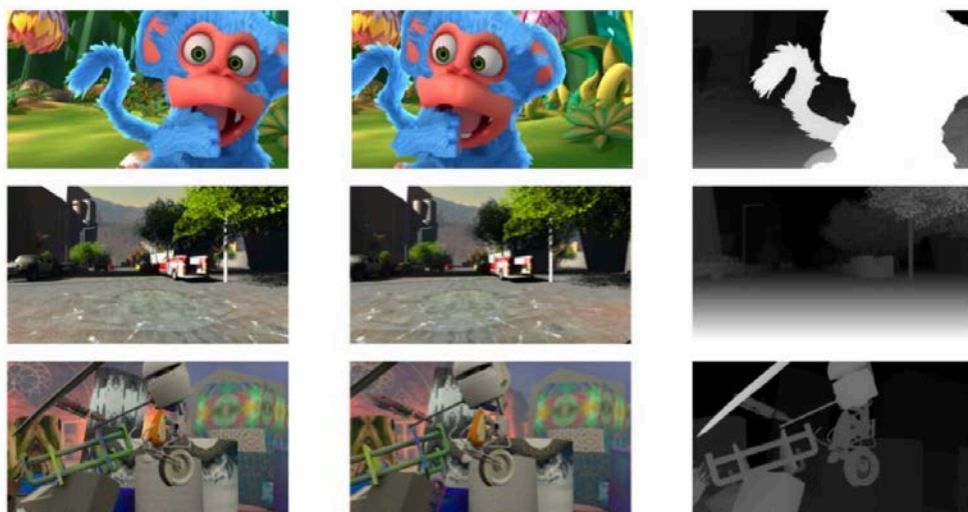


Figura 3.2. Ejemplos de pares estéreo de Scene Flow y su mapa de disparidad real.

Posteriormente, mediante fine-tuning, estos pesos se ajustan al dominio de imágenes reales proporcionado por KITTI. Esta estrategia reduce la necesidad de grandes cantidades de datos y disminuye el coste computacional respecto a un entrenamiento completo desde cero.

3.6. DATASETS UTILIZADOS

Los datasets utilizados en este trabajo son KITTI 2012 y KITTI 2015. Ambos contienen pares de imágenes estéreo capturadas desde vehículos en entornos reales y disponen de una organización basada en dos conjuntos principales: *training* y *testing*. El conjunto *training* incluye las imágenes estéreo junto con los mapas de disparidad de referencia, también denominados ground truth, mientras que el conjunto *testing* contiene las imágenes estéreo sin mapas de disparidad de referencia públicos [10].

Aunque el presente trabajo no tiene como objetivo desarrollar un sistema de conducción autónoma, KITTI resulta adecuado para estudiar la reconstrucción tridimensional del entorno mediante visión estéreo. Esto se debe a que proporciona escenas reales, pares de imágenes rectificadas, mapas de disparidad de referencia y archivos de calibración. Estos elementos permiten entrenar y evaluar modelos de estimación de disparidad, así como transformar posteriormente la disparidad estimada en información de profundidad y nubes de puntos 3D.

La organización interna de ambos datasets presenta algunas diferencias en los nombres de sus carpetas. En KITTI 2012, las imágenes estéreo del conjunto de entrenamiento se almacenan en las carpetas *colored_0* y *colored_1*, mientras que la disparidad de referencia se encuentra en *disp_occ*. En KITTI 2015, las imágenes izquierda y derecha se almacenan en *image_2* y *image_3*, y la disparidad de referencia se encuentra en *disp_occ_0*.

Además de las imágenes y los mapas de disparidad, KITTI proporciona archivos de calibración asociados a las cámaras. En el caso de KITTI 2015, estos archivos se gestionan mediante un paquete independiente de calibración. En este proyecto, la calibración se utiliza cuando es necesario obtener los parámetros de cámara, como la distancia focal y la línea base, especialmente en el cálculo de métricas por distancia y en la reconstrucción tridimensional. Para el entrenamiento con KITTI 2012 no se emplean directamente los archivos de calibración, ya que el ajuste del modelo se realiza comparando la disparidad predicha con la disparidad de referencia.

En la metodología seguida, el conjunto *training* de KITTI 2012 se utiliza para realizar el fine-tuning del modelo PSMNet. Este conjunto permite adaptar la red al dominio de imágenes reales mediante pares estéreo con sus correspondientes mapas de disparidad de referencia.

KITTI 2015 se emplea para la selección y evaluación del modelo. Dentro de su conjunto *training*, la primera mitad de las muestras se utiliza para seleccionar la mejor época durante el entrenamiento, mientras que la segunda mitad se reserva para la evaluación final y para las visualizaciones comparativas posteriores. Esta separación evita evaluar el rendimiento final sobre las mismas imágenes utilizadas para seleccionar el modelo.

Por último, el conjunto *testing* de KITTI 2015, al no disponer de mapas de disparidad de referencia públicos, se utiliza en la fase final del proyecto para realizar inferencia y generar reconstrucciones 3D a partir de pares estéreo. En este caso no se calculan métricas cuantitativas frente a ground truth, sino que se emplean las predicciones de PSMNet junto con los parámetros de calibración para obtener mapas de profundidad y nubes de puntos tridimensionales.

3.7. MÉTRICAS UTILIZADAS

Para cuantificar la calidad de los mapas de disparidad obtenidos se comparan las predicciones con la disparidad real proporcionada por el dataset. El conjunto de píxeles válidos se define como aquellos con disparidad real positiva y menor que el valor máximo de disparidad considerado.

Teniendo en cuenta que:

$$\Omega = \left\{ i \mid d_i^{gt} > 0 \text{ AND } d_i^{gt} < d_{max} \right\}$$

$$N = |\Omega|$$

Donde:

Ω = conjunto de píxeles válidos

N = número de píxeles válidos

d^{gt} = disparidad real

d^{pred} = disparidad predicha

La métrica EPE, o End Point Error, mide el error absoluto medio entre la disparidad predicha y la disparidad real:

$$EPE = \frac{1}{N} \sum_{i \in \Omega} |d_i^{pred} - d_i^{gt}|$$

La métrica AbsRel, o error relativo absoluto, calcula la media del error absoluto dividido entre la disparidad real:

$$AbsRel = \frac{1}{N} \sum_{i \in \Omega} \frac{|d_i^{pred} - d_i^{gt}|}{d_i^{gt}}$$

La métrica delta1 mide el porcentaje medio de píxeles cuya predicción se encuentra dentro de un umbral relativo respecto al valor real. En este trabajo se considera correcta una predicción cuando la relación entre la disparidad estimada y la real es inferior a 1,25:

$$\delta_1 = \frac{1}{N} \sum_{i \in \Omega} 1 \left(\max \left(\frac{d_i^{pred}}{d_i^{gt}}, \frac{d_i^{gt}}{d_i^{pred}} \right) < 1.25 \right)$$

Por último, D1-all es una métrica utilizada en el benchmark KITTI para medir el porcentaje medio de píxeles considerados erróneos. Según este criterio, un píxel se considera erróneo cuando el error absoluto es superior a 3 píxeles y, además, supera el 5 % de la disparidad real:

$$D1_{all} = \frac{1}{N} \sum_{i \in \Omega} 1 \left(\left| d_i^{pred} - d_i^{gt} \right| > 3 \text{ AND } \frac{\left| d_i^{pred} - d_i^{gt} \right|}{d_i^{gt}} > 0.05 \right)$$

El uso conjunto de estas métricas permite evaluar el rendimiento desde distintos puntos de vista: EPE resume el error absoluto independientemente de que la disparidad predicha se haya estimado errónea a la izquierda o a la derecha de la disparidad real, D1-all mide el porcentaje de errores significativos, AbsRel introduce una interpretación relativa del error y delta1 da por correcta una predicción mientras su error no supere el 25%.

3.8. ESQUEMA GENERAL DE LA METODOLOGÍA EXPERIMENTAL

La *Figura 3.3* resume el flujo experimental seguido en este trabajo. A partir de un modelo PSMNet preentrenado en Scene Flow se realizan distintos experimentos de fine-tuning, se evalúan los modelos obtenidos, se comparan con un método clásico de reconstrucción estéreo y finalmente se selecciona el modelo más adecuado para generar las reconstrucciones 3D finales.

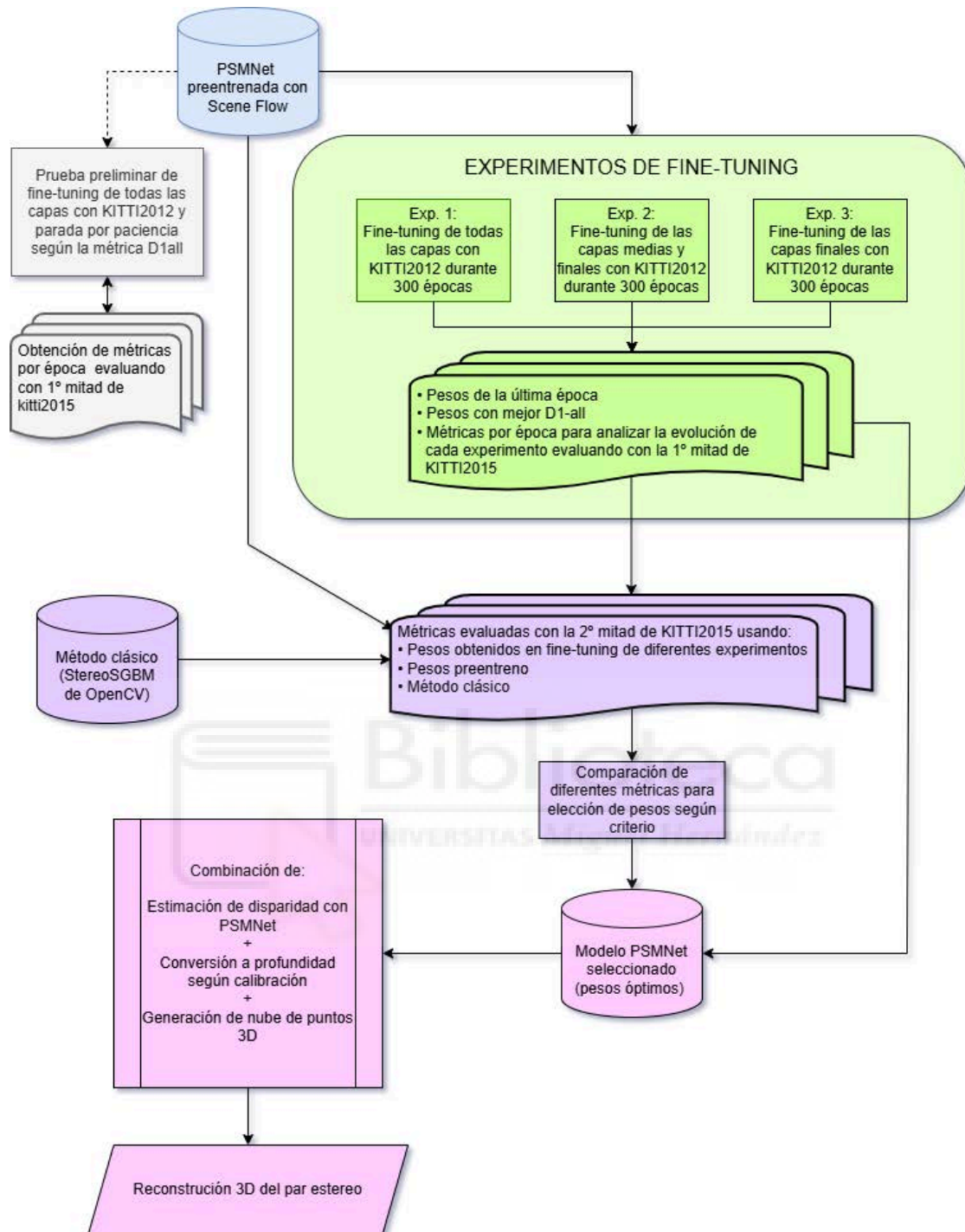


Figura 3.3. Esquema general de la metodología experimental.

4. RESULTADOS Y DISCUSIÓN

4.1. MÉTODO CLÁSICO DE COMPARACIÓN

Además del modelo basado en PSMNet, he implementado un método clásico de correspondencia estéreo con el objetivo de disponer de una referencia comparativa, adjuntado el código en el *ANEXO 1*. Para ello se utiliza StereoSGBM [4] de OpenCV, una implementación basada en Semi-Global Matching que estima la disparidad a partir de la comparación densa entre la imagen izquierda y la imagen derecha.

Este método no emplea aprendizaje profundo ni entrenamiento previo. Su inclusión permite comparar el comportamiento de una técnica tradicional frente a una arquitectura basada en CNN bajo el mismo conjunto de evaluación y con las mismas métricas principales.

Experimentalmente, ejecutando este método y evaluando con la 2º mitad del dataset de KITTI2015, se han obtenido las siguientes métricas:

Métricas método geométrico	
D1-all	21.412%
EPE	7.766118
Abs Rel	22.491%
delta1	79.612%
predicción de mapas de disparidad por segundo	16.863872

Para un análisis visual, también se realizan mapas de disparidad predicha de todas las muestras, aquí un ejemplo comparándose con la imagen izquierda del par y su disparidad real:

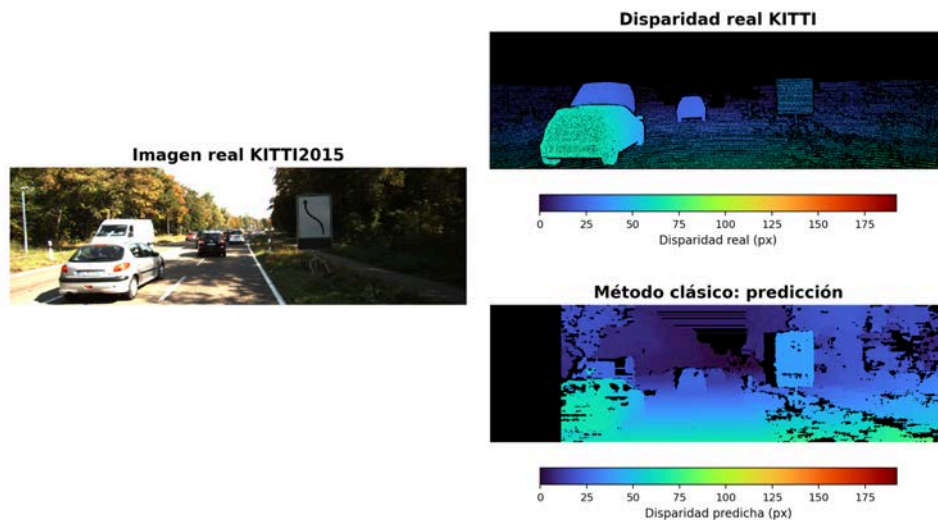


Figura 4.1. Ejemplo comparación entre disparidad real y su predicción.

En las predicciones obtenidas mediante StereoSGBM se observa una franja negra en el borde izquierdo de la imagen. Esta zona no corresponde a un error de implementación, sino a una limitación habitual de los métodos clásicos de correspondencia estéreo. Al buscar correspondencias en un rango de disparidades positivo, los píxeles situados cerca del borde izquierdo no disponen de suficiente margen para evaluar todo el intervalo de búsqueda, por lo que el algoritmo devuelve valores inválidos o nulos. En las visualizaciones estos valores se representan en negro. No obstante, dichos píxeles no se eliminan del cálculo de métricas cuando el ground truth es válido, por lo que el efecto queda penalizado en la evaluación cuantitativa.

4.2. FINE-TUNING Y EXPERIMENTOS DE ENTRENAMIENTO

4.2.1. ITERACIONES DE ENTRENAMIENTO Y ÉPOCAS

Antes de describir el proceso de entrenamiento, conviene aclarar la diferencia entre los conceptos de época, lote e iteración. Una época, o epoch, es una pasada completa del conjunto de datos de entrenamiento a través de la red neuronal. Es decir, durante una época el modelo procesa todas las muestras disponibles para entrenamiento y actualiza sus pesos a partir del error cometido en la predicción.

Como normalmente no es posible procesar todas las imágenes del dataset al mismo tiempo por limitaciones de memoria, el conjunto de entrenamiento se divide en lotes o batches. Cada lote contiene un número reducido de muestras

que se procesan de forma conjunta. Una iteración corresponde al procesamiento de uno de estos lotes y a la actualización de los pesos de la red a partir de él. Por tanto, una época está formada por tantas iteraciones como lotes sean necesarios para recorrer el conjunto de entrenamiento completo.

Una muestra, para nuestro caso, se considera como un par estéreo formado por una imagen izquierda, una imagen derecha y su correspondiente mapa de disparidad real.

Por ejemplo, si se dispone de 100 muestras y se utiliza un tamaño de lote de 4 imágenes, una época estaría formada por 25 iteraciones. Si el tamaño de lote fuese 1, esa misma época estaría formada por 100 iteraciones. En ambos casos, la época representa una pasada completa por el conjunto de datos; lo que cambia es el número de actualizaciones intermedias realizadas durante dicha pasada.

En este trabajo, el entrenamiento se realiza utilizando KITTI 2012 como conjunto de ajuste, con un tamaño de lote arbitrario de 4 muestras durante el fine-tuning. Dado que KITTI 2012 contiene 194 muestras de entrenamiento, cada época se compone de 49 iteraciones: 48 lotes completos de 4 muestras y un último lote de 2 muestras, ya que no se descarta el lote final incompleto.

Entrenar durante varias épocas permite que la red neuronal ajuste progresivamente sus pesos y mejore su capacidad para estimar mapas de disparidad. Sin embargo, aumentar el número de épocas no implica necesariamente obtener siempre mejores resultados. A partir de cierto punto puede aparecer sobreajuste, u overfitting, situación en la que el modelo se adapta demasiado a los datos utilizados durante el entrenamiento y pierde capacidad de generalización sobre nuevas imágenes. Por este motivo, resulta necesario evaluar el modelo tras cada época y seleccionar aquella que ofrezca el mejor rendimiento sobre un conjunto de validación independiente.

Por ello, tras cada época el modelo se evalúa sobre la primera mitad de KITTI 2015, empleando métricas cuantitativas de error de disparidad. En concreto, se utiliza D1-all como criterio principal para determinar la mejor época, siguiendo el criterio empleado habitualmente en el benchmark KITTI.

4.2.2. EXPERIMENTO PREVIO CON EARLY STOPPING

Una primera estrategia fue hacerlo con el método de Parada Temprana, o Early Stopping, que consiste en parar el entrenamiento si después de un número de épocas, llamada paciencia, no ha mejorado la métrica de seguimiento.

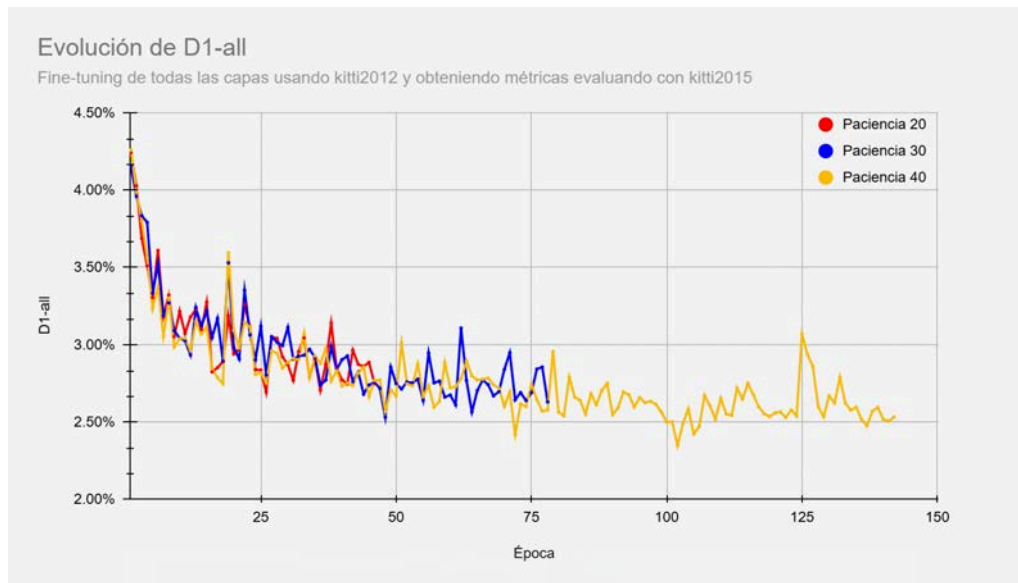


Figura 4.2. Evolución de D1-all durante fine-tuning con diferentes paciencias.

Pero este método de Parada Temprana fue descartado. En su lugar preferí hacerlo a 300 épocas fijas, como recomienda el repositorio oficial de PSMNet, para así poder comparar mejor entre diferentes experimentos de entrenamiento por capas bajo las mismas condiciones de número de épocas máximas.

4.2.3. FINE-TUNING EN DIFERENTES CAPAS

Durante el fine-tuning se han planteado diferentes configuraciones para analizar cómo afecta el grado de actualización de la red al rendimiento final. En concreto, se consideran tres experimentos: entrenamiento de todas las capas, entrenamiento de capas intermedias y finales, y entrenamiento de las capas finales.

Para los tres experimentos, además de guardar los pesos de entrenamiento de la última época, se han guardado los pesos de la época con menor D1-all antes de la última época, para en caso de observar sobreajuste quedarnos con los

pesos óptimos. En el *ANEXO 2* se adjunta el código completo de entrenamiento y seguimiento por métricas. A continuación se presentan los resultados obtenidos en el proceso de fine-tuning graficados, tanto para D1-all como para otras métricas para observar la correlación de los resultados. También se acompaña de una tabla resumen, donde se ha marcado en verde la celda con el valor óptimo según la métrica.

D1-all

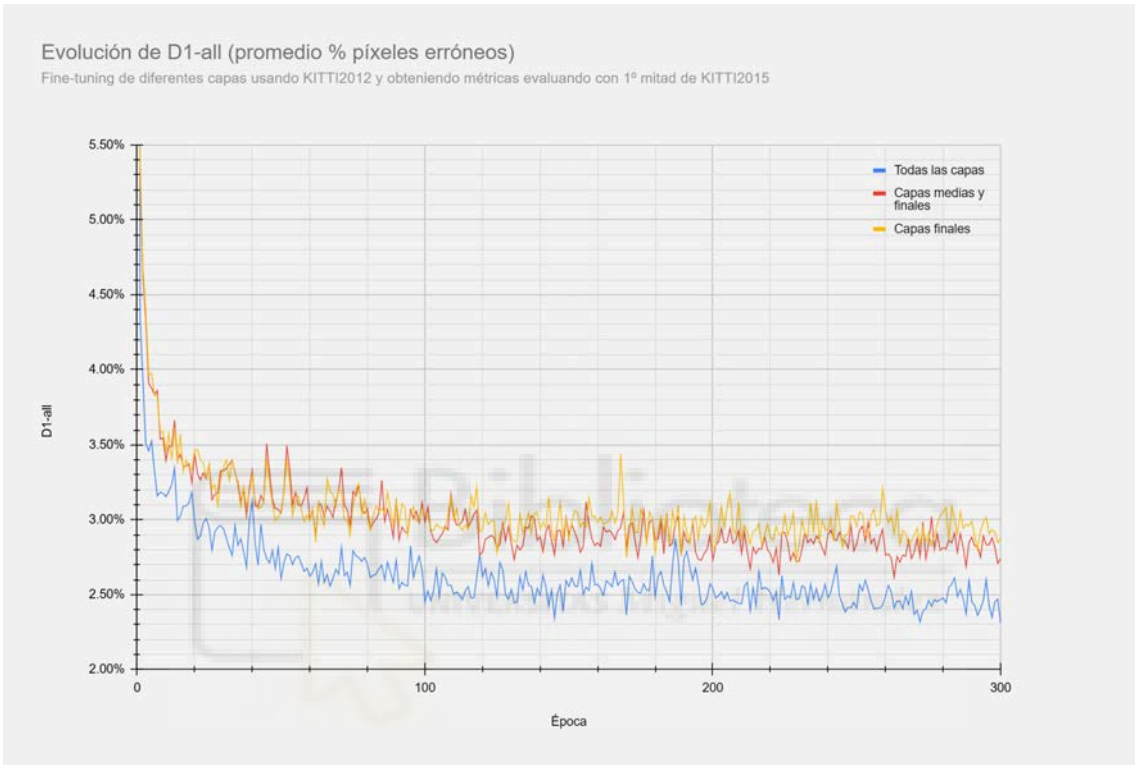


Figura 4.3. Evolución de D1-all durante fine-tuning de diferentes capas.

		D1-all en fine-tuning		
	Época	Todas las capas	Capas medias y finales	Capas finales
Mínimo antes de la última época	229			2.7195%
	263		2.6161%	
	272	2.3183%		
Última época	300	2.3133%	2.7410%	2.8791%

EPE

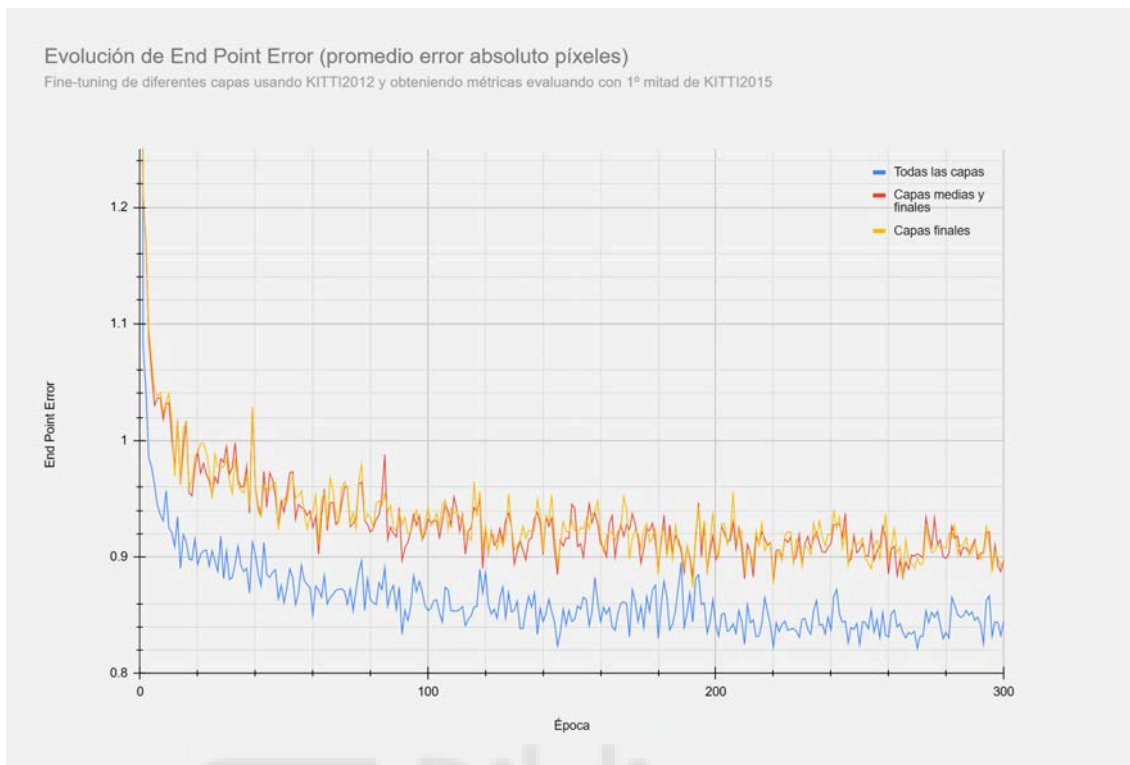


Figura 4.4. Evolución de EPE durante fine-tuning de diferentes capas.

End Point Error en fine-tuning				
	Época	Todas las capas	Capas medias y finales	Capas finales
Mínimo antes de la última época	192			0.873615
	220		0.879058	
	270	0.821899		
Última época	300	0.844828	0.897353	0.896218

AbsRel

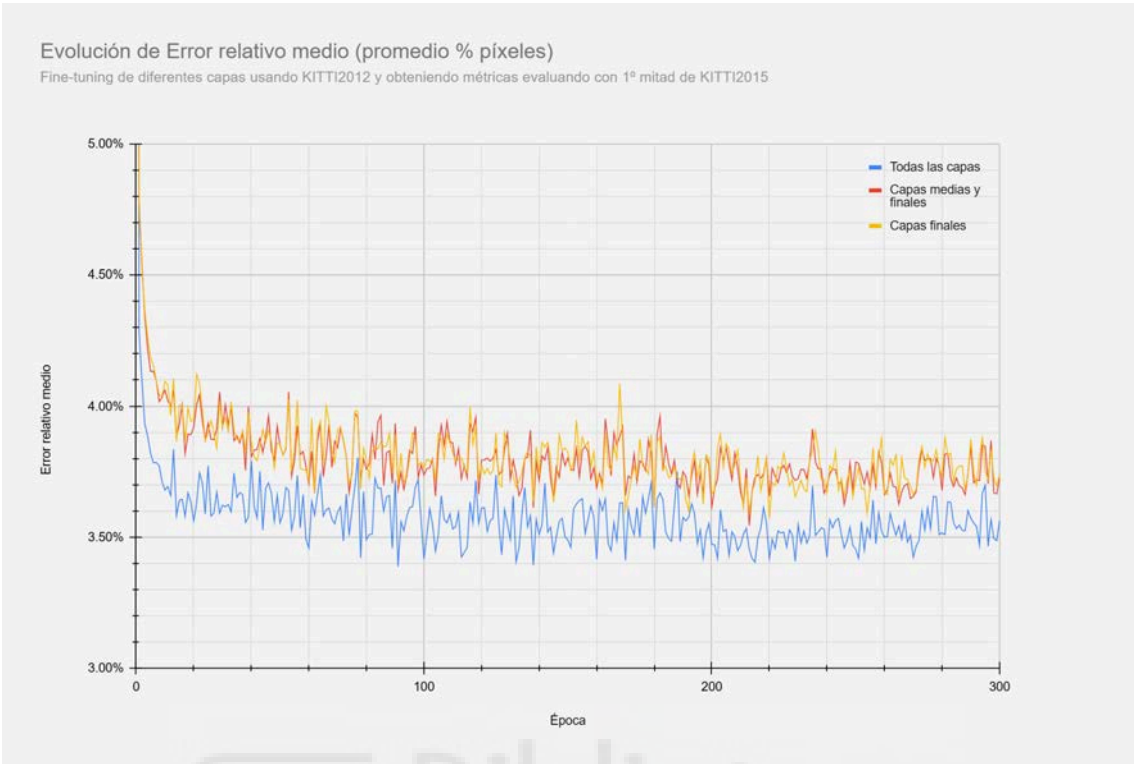


Figura 4.5. Evolución de Abs Rel durante fine-tuning de diferentes capas.

		Error relativo medio en fine-tuning		
	Época	Todas las capas	Capas medias y finales	Capas finales
Mínimo antes de la última época	91	3.3900%		
	213		3.5461%	
	220			3.5744%
Última época	300	3.5643%	3.7251%	3.7417%

Delta1

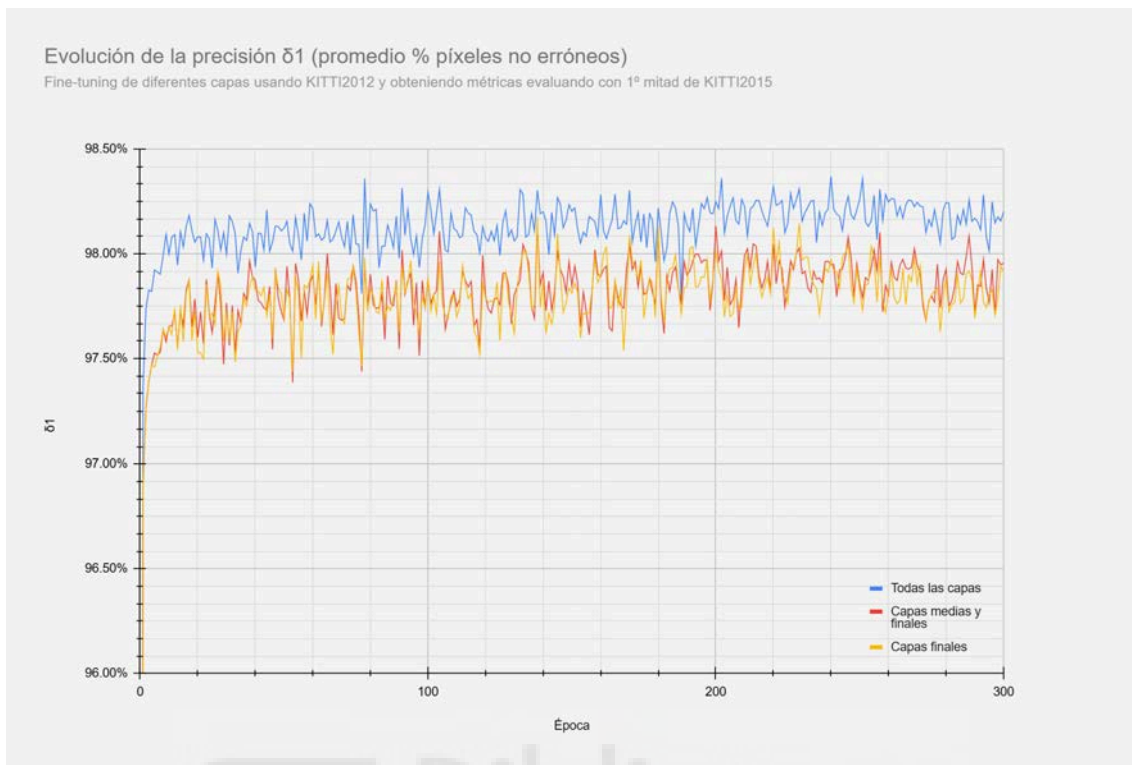


Figura 4.6. Evolución de delta1 durante fine-tuning de diferentes capas.

		$\delta 1$ en fine-tuning		
	Época	Todas las capas	Capas medias y finales	Capas finales
Máximo antes de la última época	138			98.1739%
	138		98.1340%	
	240	98.3691%		
Última época	300	98.2008%	97.9555%	97.9130%

4.3. EVALUACIÓN

4.3.1. MÉTRICAS PRINCIPALES

Una vez finalizado el proceso de fine-tuning y seleccionadas las épocas más representativas de cada experimento, se realiza una evaluación final empleando la segunda mitad de KITTI2015. Esta división se utilizó para disponer de un conjunto de prueba independiente, distinto al empleado durante la selección de la, presuntamente, mejor época. De este modo, se evita que la elección de los pesos esté condicionada por las mismas imágenes usadas para la evaluación final y se obtiene una comparación más objetiva entre los distintos modelos.

También se evalúa con los pesos de preentrenamiento en el dataset Scene Flow, para tener una referencia del punto de partida, pudiendo considerarse esta la época 0. En las siguientes gráficas se muestran las métricas a partir de la época 1 para poder mantener una escala visual más fácil de observar.

El código utilizado para esta parte se adjunta en el *ANEXO 3*.

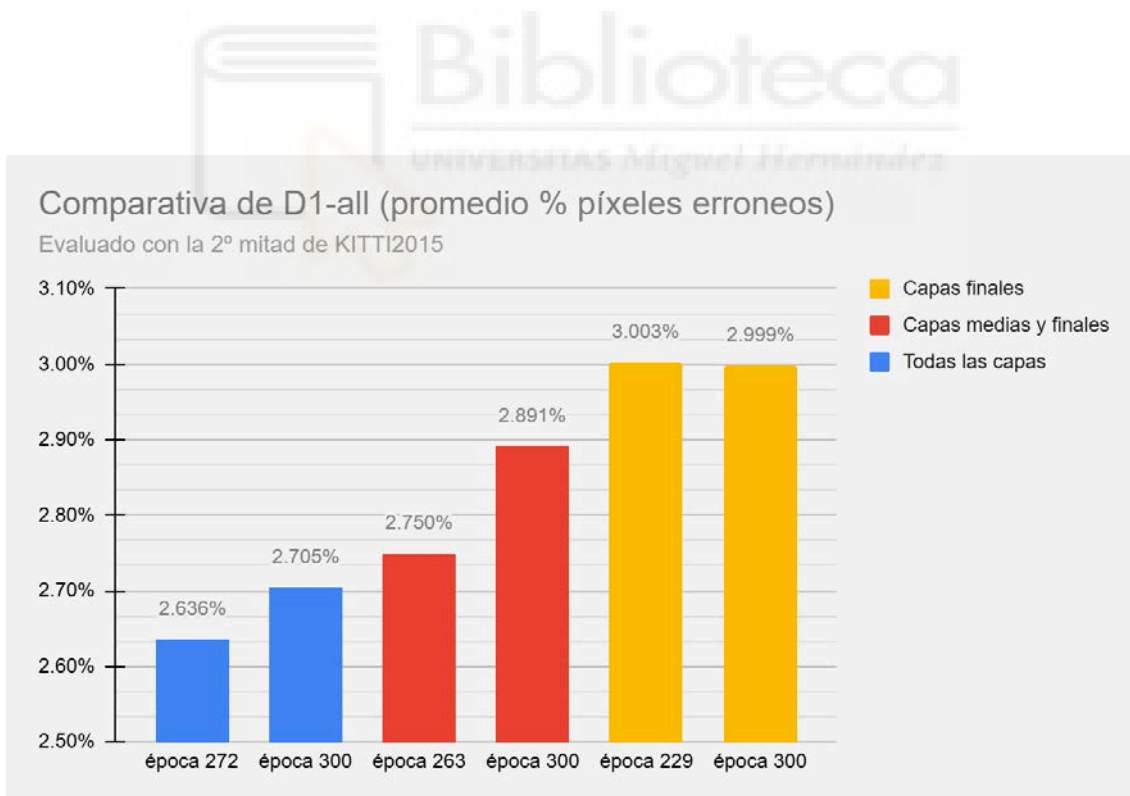


Figura 4.7. Comparativa de D1-all en evaluación de métricas.

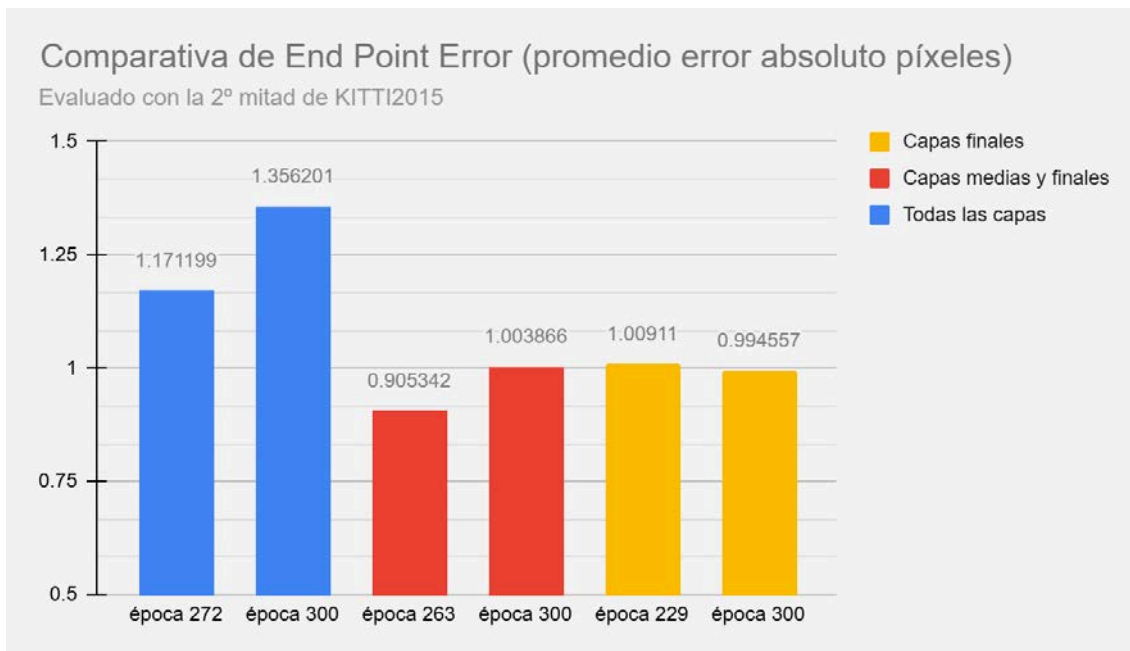


Figura 4.8. Comparativa de EPE en evaluación de métricas.

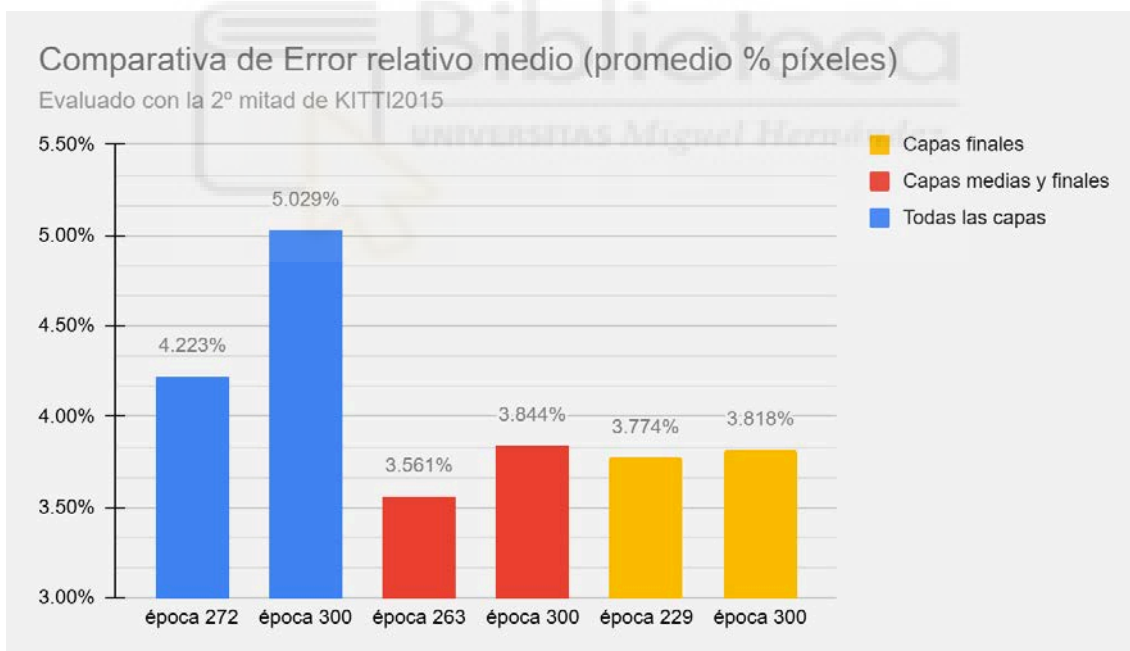


Figura 4.9. Comparativa de Abs Rel en evaluación de métricas.

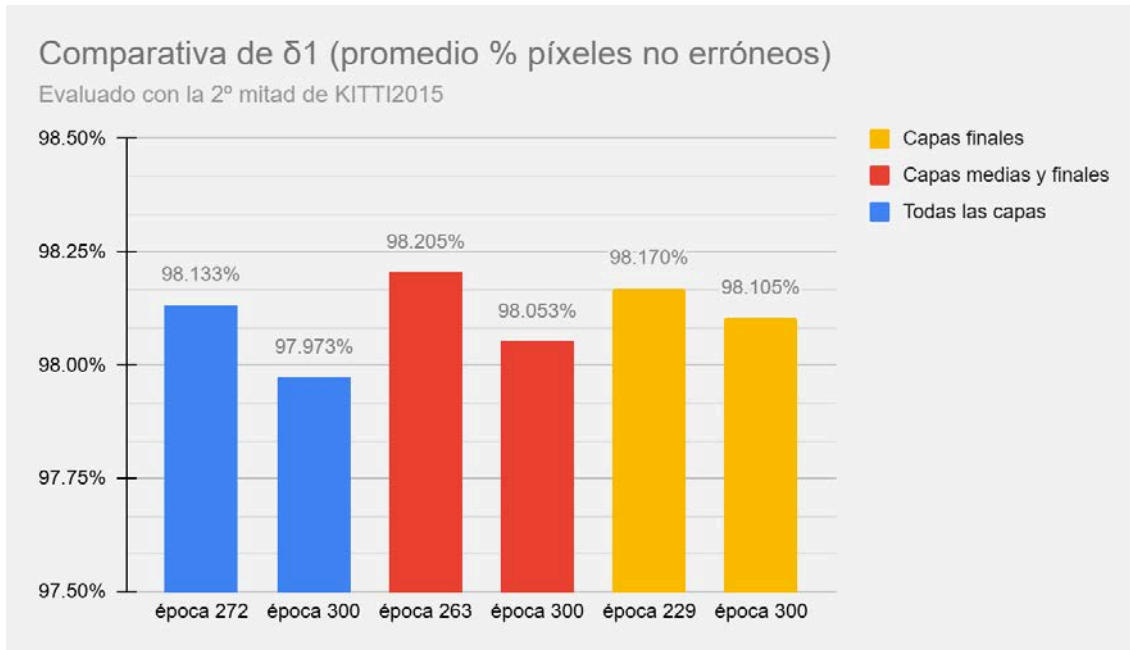


Figura 4.10. Comparativa de delta1 en evaluación de métricas.

A continuación, se muestra una tabla resumen para ver esta última evaluación de forma global, donde las celdas en verde y rojo muestran el mejor y peor resultado, respectivamente, de cada métrica. También se incluye la diferencia entre el mejor y peor resultado por cada métrica.

		Evaluación 2º mitad KITTI2015			
Pesos entrenamiento	Época	D1-all	EPE	Abs Rel	$\delta 1$
Preentrenado	0	28.692%	4.755336	17.190%	88.175%
Todas las capas	272	2.636%	1.171199	4.223%	98.133%
	300	2.705%	1.356201	5.029%	97.973%
Capas medias y finales	263	2.750%	0.905342	3.561%	98.205%
	300	2.891%	1.003866	3.844%	98.053%
Capas finales	229	3.003%	1.009110	3.774%	98.170%
	300	2.999%	0.994557	3.818%	98.105%
Diferencia entre mejor y peor resultado		0.366%	0.450859	1.468%	0.232%

Al comparar los pesos obtenidos en la época 272 del entrenamiento de todas las capas con los obtenidos en la época 263 del entrenamiento de capas medias y finales, se observa que ambas configuraciones presentan resultados muy competitivos, aunque con comportamientos ligeramente diferentes.

La configuración de todas las capas en la época 272 obtiene el mejor valor de D1-all, con un 2.636 %. Esto indica que es la opción que produce un menor porcentaje de píxeles considerados erróneos según el criterio del benchmark KITTI. Por tanto, esta alternativa resulta interesante cuando se desea optimizar directamente la métrica principal de KITTI o reducir el número de errores grandes de disparidad. Sin embargo, esta configuración presenta valores peores de EPE y AbsRel que la configuración de capas medias y finales, lo que indica que, aunque comete ligeramente menos errores graves, el error medio de disparidad es mayor.

Por otro lado, la configuración de capas medias y finales en la época 263 obtiene mejores resultados en EPE, AbsRel y $\delta 1$. Esto significa que, en promedio, sus predicciones de disparidad se aproximan más a la disparidad real, aunque el porcentaje de píxeles clasificados como erróneos según D1-all sea ligeramente superior. Además, al no reajustar todas las capas de la red, esta configuración conserva mejor las características generales aprendidas durante el preentrenamiento y adapta principalmente las capas más relacionadas con la estimación final de disparidad. Esto puede favorecer una mayor estabilidad y reducir el riesgo de sobreajuste.

Por tanto, si el criterio principal es obtener el mejor valor posible en D1-all, la opción más adecuada sería utilizar los pesos de todas las capas en la época 272. En cambio, si se busca una reconstrucción más precisa en términos de error medio y error relativo, la opción de capas medias y finales en la época 263 resulta más equilibrada. Esta última puede ser más adecuada para aplicaciones de reconstrucción 3D densa, generación de nubes de puntos o análisis geométrico, donde interesa que la disparidad media sea lo más precisa posible. La opción de todas las capas puede ser preferible en aplicaciones donde se priorice reducir el número de píxeles con errores grandes, como comparaciones directas con benchmarks o sistemas donde los outliers sean especialmente críticos como por ejemplo conducción autónoma.

4.3.2. ANÁLISIS DE SOBREAJUSTE

Para analizar la posible aparición de sobreajuste, se ha reorganizado la tabla anterior para comparar las métricas obtenidas en la mejor época seleccionada de cada experimento por separado (según D1-all en fine-tuning evaluando con la 1ª mitad de KITTI2015) con las correspondientes a la última época de entrenamiento, la época 300. También se incluye la diferencia entre la época 300 y su anterior correspondiente según el experimento.

		Evaluación 2º mitad KITTI2015			
Pesos entrenamiento	Época	D1-all	EPE	Abs Rel	$\delta 1$
Todas las capas	272	2.636%	1.171199	4.223%	98.133%
	300	2.705%	1.356201	5.029%	97.973%
Diferencia entre épocas		0.069%	0.185002	0.805%	-0.160%

		Evaluación 2º mitad KITTI2015			
Pesos entrenamiento	Época	D1-all	EPE	Abs Rel	$\delta 1$
Capas medias y finales	263	2.750%	0.905342	3.561%	98.205%
	300	2.891%	1.003866	3.844%	98.053%
Diferencia entre épocas		0.142%	0.098524	0.284%	-0.153%

		Evaluación 2º mitad KITTI2015			
Pesos entrenamiento	Época	D1-all	EPE	Abs Rel	$\delta 2$
Capas finales	229	3.003%	1.009110	3.774%	98.170%
	300	2.999%	0.994557	3.818%	98.105%
Diferencia entre épocas		-0.004%	-0.014553	0.044%	-0.065%

En el experimento de entrenamiento de todas las capas se observa un empeoramiento al pasar de la época 272 a la 300: D1-all aumenta de 2.636 % a 2.705 %, EPE pasa de 1.171199 a 1.356201 y AbsRel aumenta de 4.223 % a 5.029 %. Además, $\delta 1$ disminuye ligeramente. Este comportamiento indica que

continuar el entrenamiento después de la mejor época no mejora la capacidad de generalización del modelo y puede asociarse a un inicio de sobreajuste.

Un comportamiento similar se aprecia en el experimento de capas medias y finales. La época 263 obtiene mejores resultados que la época 300 en todas las métricas principales, especialmente en EPE y AbsRel. Esto refuerza la decisión de seleccionar la mejor época según el conjunto de validación, en lugar de utilizar directamente la última época entrenada.

En el caso de las capas finales, las diferencias entre la época 229 y la 300 son mucho menores. Algunas métricas mejoran ligeramente y otras empeoran, por lo que no se observa un sobreajuste claro, sino un comportamiento más estable. En conjunto, estos resultados justifican el uso de la mejor época validada para cada experimento, ya que prolongar el entrenamiento hasta la época final no garantiza mejores resultados en datos no usados durante el ajuste.

4.3.3. OTRAS MÉTRICAS

Métricas en función de la distancia

Además de las métricas globales, se calcularon las mismas métricas pero agrupadas por rango de distancia. Estas métricas no se emplearon como criterio principal para seleccionar el modelo final, sino como análisis complementario para estudiar el comportamiento de la red en distintas zonas de profundidad de la escena. De este modo, se puede observar si el ajuste fino afecta de forma diferente a objetos cercanos, a distancias medias o a regiones más alejadas.

Para ello, la distancia se calcula a partir de la disparidad real de KITTI, junto con los datos de calibración, y se utilizó únicamente para clasificar los píxeles en rangos de distancia. Una vez definidos estos grupos, los errores se calcularon sobre la disparidad, comparando la disparidad predicha por la red con la disparidad real. Por tanto, estas tablas no representan un error directo de profundidad en metros, sino métricas de disparidad separadas según la distancia real estimada de cada píxel.

Comparación de D1-all por rango de distancia								
Rango distancia	Preentrenado	Todas las capas época 272	Todas las capas época 300	Capas medias y finales época 263	Capas medias y finales época 300	Capas finales: época 229	Capas finales: época 300	Diferencia entre mejor y peor resultado
0 a 30 m	30.07%	2.617%	2.684%	2.728%	2.860%	2.969%	2.994%	0.377%
31 a 60 m	17.79%	3.037%	3.077%	3.262%	3.447%	3.567%	3.325%	0.529%
61 a 90 m	15.86%	3.642%	3.997%	3.912%	4.165%	4.964%	4.079%	1.322%

Comparación de EPE por rango de distancia								
Rango distancia	Preentrenado	Todas las capas época 272	Todas las capas época 300	Capas medias y finales época 263	Capas medias y finales época 300	Capas finales: época 229	Capas finales: época 300	Diferencia entre mejor y peor resultado
0 a 30 m	4.856286	1.179371	1.356359	0.899115	1.003365	1.019847	0.997329	0.457244
31 a 60 m	4.152675	0.999986	1.40414	1.030603	1.020865	0.953336	1.005094	0.450804
61 a 90 m	3.49194	0.958705	1.223345	0.871953	0.867723	0.927711	0.861546	0.361799

Comparación de AbsRel por rango de distancia								
Rango distancia	Preentrenado	Todas las capas época 272	Todas las capas época 300	Capas medias y finales época 263	Capas medias y finales época 300	Capas finales: época 229	Capas finales: época 300	Diferencia entre mejor y peor resultado
0 a 30 m	14.827%	3.529%	4.191%	2.787%	3.096%	3.083%	3.093%	1.404%
31 a 60 m	44.397%	10.628%	14.668%	10.862%	10.784%	10.082%	10.533%	4.586%
61 a 90 m	59.478%	16.410%	20.780%	15.064%	14.980%	16.053%	14.901%	5.879%

Comparación de ̂1 por rango de distancia								
Rango distancia	Preentrenado	Todas las capas época 272	Todas las capas época 300	Capas medias y finales época 263	Capas medias y finales época 300	Capas finales: época 229	Capas finales: época 300	Diferencia entre mejor y peor resultado
0 a 30 m	89.678%	98.543%	98.396%	98.746%	98.583%	98.646%	98.602%	0.350%
31 a 60 m	75.889%	94.197%	93.984%	92.666%	92.833%	93.597%	93.170%	1.531%
61 a 90 m	68.422%	93.971%	92.918%	92.098%	92.129%	90.417%	92.053%	3.555%

En general, los resultados muestran que el fine-tuning mejora claramente el comportamiento del modelo respecto al preentrenado en los distintos rangos analizados. También se observa que, a medida que aumenta la distancia, los errores tienden a ser mayores. Esto puede deberse a que los objetos lejanos presentan disparidades más pequeñas, ocupan menos píxeles en la imagen y ofrecen menos información visual para establecer correspondencias precisas entre ambas cámaras. Por ello, estas métricas permiten comprobar de forma complementaria cómo responde la CNN ante distintas profundidades de la escena.

Métrica temporales

También se registró el tiempo medio de inferencia necesario para calcular el mapa de disparidad. Esta métrica se incluyó como información complementaria, no como criterio principal de selección de pesos, ya que todos los modelos evaluados mantienen la misma arquitectura PSMNet y, durante la evaluación, se ejecuta la red completa.

	Tiempo medio de inferencia para calcular la disparidad							Diferencia entre mejor y peor resultado
	Preentrenado	Todas las capas época 272	Todas las capas época 300	Capas medias y finales época 263	Capas medias y finales época 300	Capas finales: época 229	Capas finales: época 300	
segundos por par estéreo	336.27E-3	309.67E-3	311.61E-3	310.17E-3	310.86E-3	309.61E-3	310.97E-3	1.99E-3
pares estéreo por segundo	2.9738	3.2293	3.2092	3.2240	3.2169	3.2298	3.2157	0.0207

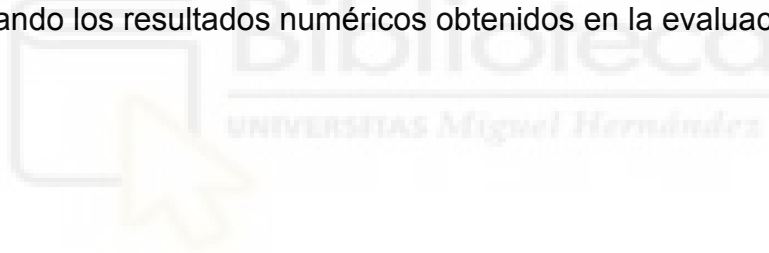
Los resultados muestran que las diferencias de tiempo entre los modelos ajustados son muy reducidas. Esto es coherente, ya que congelar capas afecta principalmente al proceso de entrenamiento, pero no reduce de forma significativa el coste de inferencia si se sigue utilizando la arquitectura completa. Por tanto, las pequeñas variaciones observadas pueden atribuirse en gran medida a la variabilidad propia de la ejecución en GPU y no justifican por sí solas la elección de un modelo frente a otro.

En consecuencia, la selección final de pesos debe basarse principalmente en las métricas de precisión, mientras que el tiempo de inferencia sirve para comprobar que todos los modelos presentan un comportamiento temporal similar.

4.4. COMPARACIÓN DE MAPAS DE DISPARIDAD

Para el modelo final se seleccionaron los pesos correspondientes a la época 263 del experimento de ajuste de capas intermedias y finales. Aunque la época 272 presentaba un resultado competitivo, la elección de la época 263 se realizó atendiendo al comportamiento global del modelo y no únicamente a una métrica aislada. En este caso, se priorizó el equilibrio entre las distintas métricas de evaluación y la calidad visual de las predicciones, evitando escoger una época únicamente por una mejora puntual en D1-all. De este modo, los pesos seleccionados representan una solución más estable y adecuada para la reconstrucción 3D final.

En el *ANEXO 4* se incluye el código empleado para la comparación visual entre el método clásico y la CNN entrenada. Este código carga varias muestras de conjunto training de KITTI2015, obtiene la imagen RGB original, la disparidad real, la predicción generada por el método clásico y la predicción obtenida mediante PSMnet. A partir de estos datos genera figuras comparativas en las que se muestran las predicciones de ambos métodos junto con sus mapas de error respecto a la disparidad real. La finalidad principal de este código es facilitar una comparación visual directa entre ambas aproximaciones, complementando los resultados numéricos obtenidos en la evaluación.



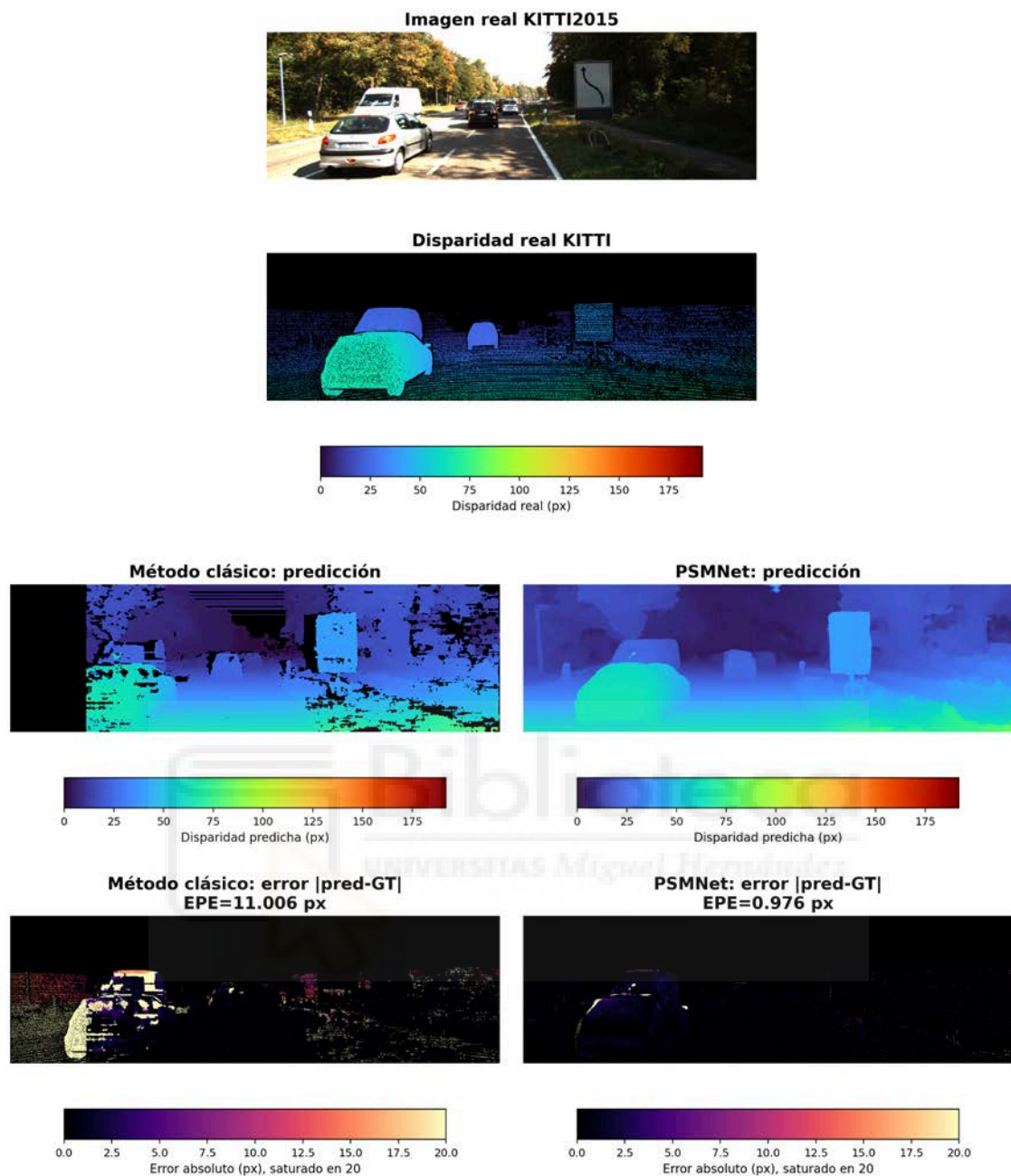


Figura 4.11. Comparación de disparidad y error entre el método clásico y PSMNet en la muestra 000163_10 de KITTI2015.

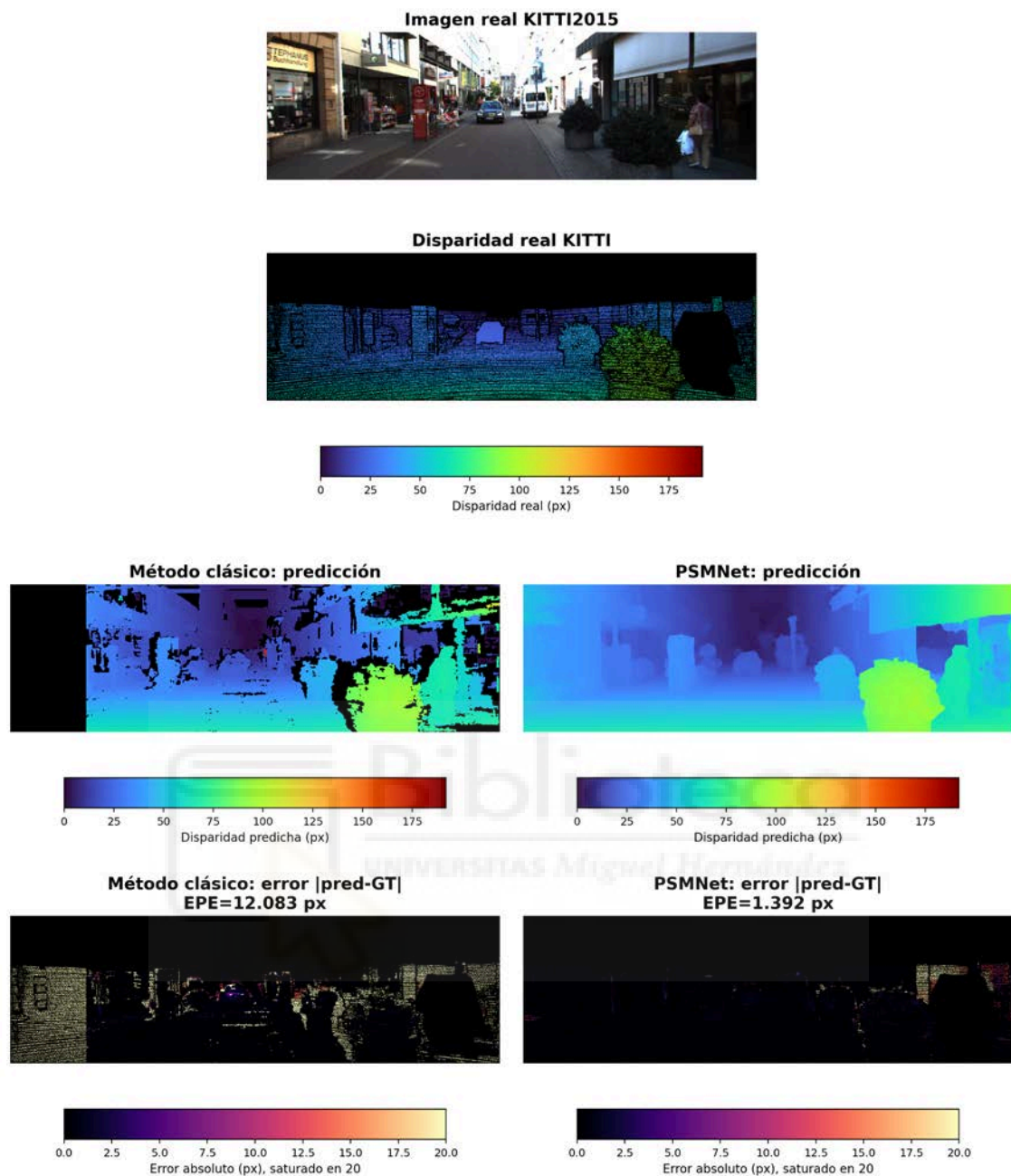


Figura 4.12. Comparación de disparidad y error entre el método clásico y PSMNet en la muestra 000168_10 de KITTI2015.

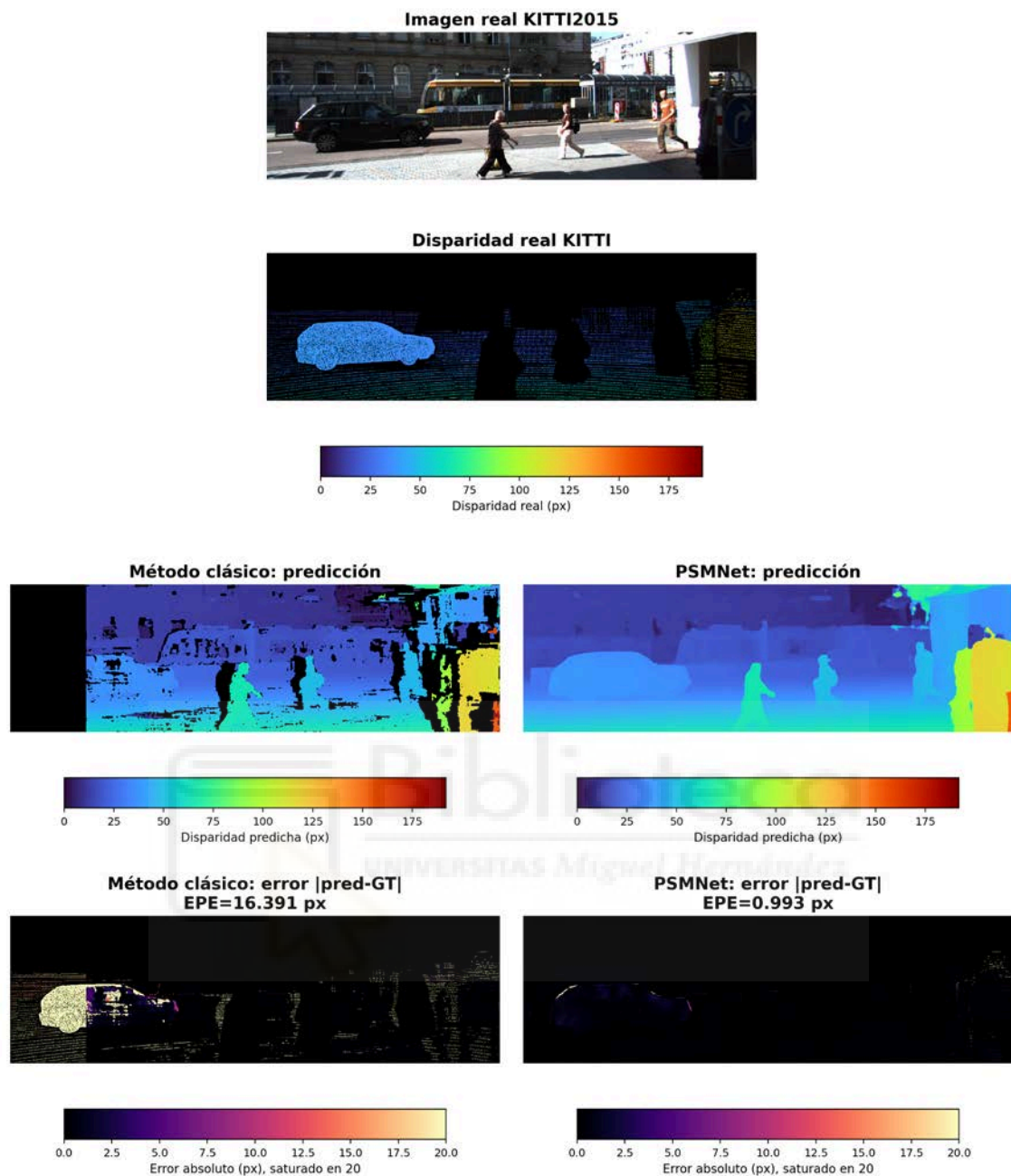


Figura 4.13. Comparación de disparidad y error entre el método clásico y PSMNet en la muestra 000169_10 de KITTI2015.

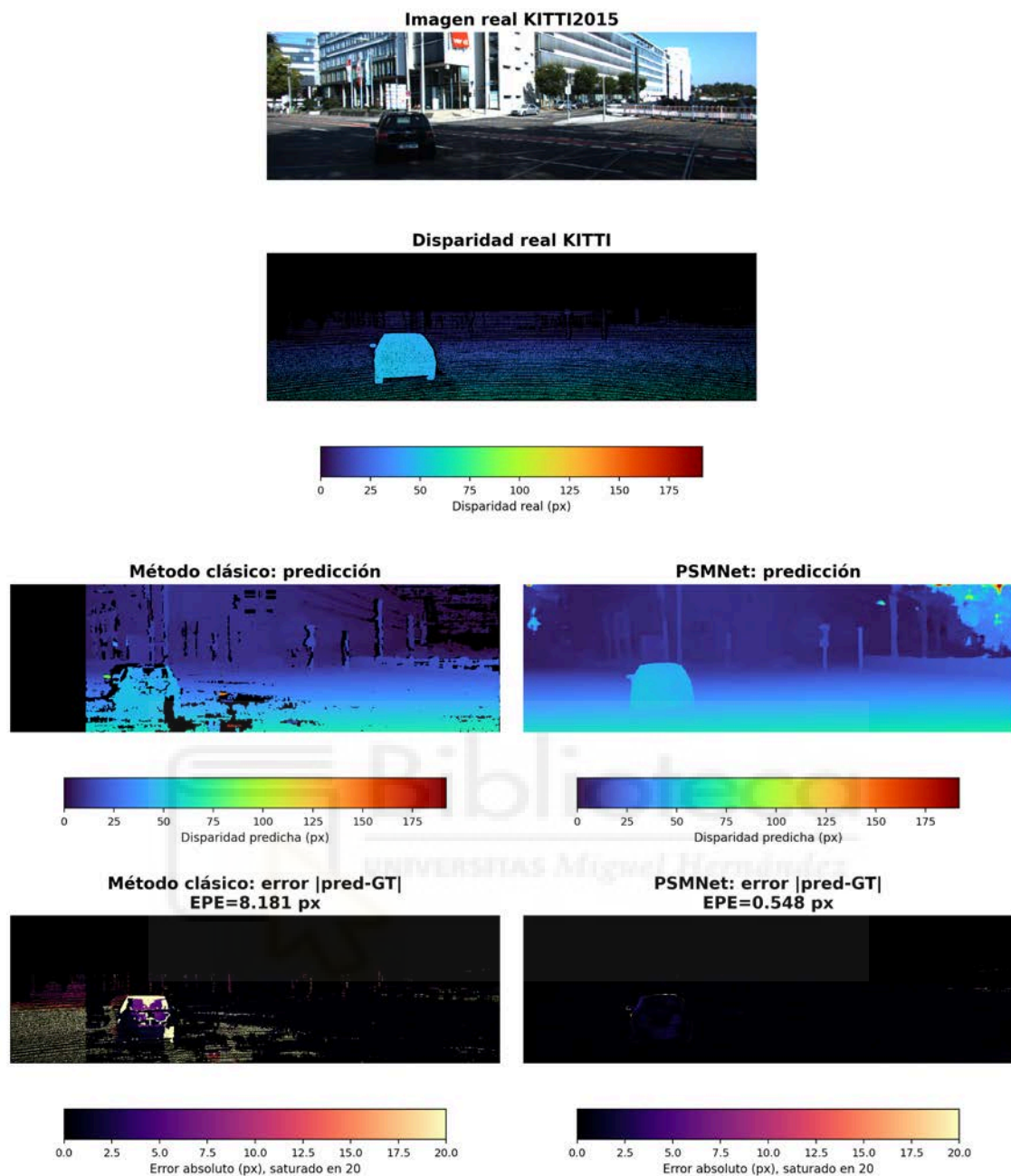


Figura 4.14. Comparación de disparidad y error entre el método clásico y PSMNet en la muestra 000171_10 de KITTI2015.

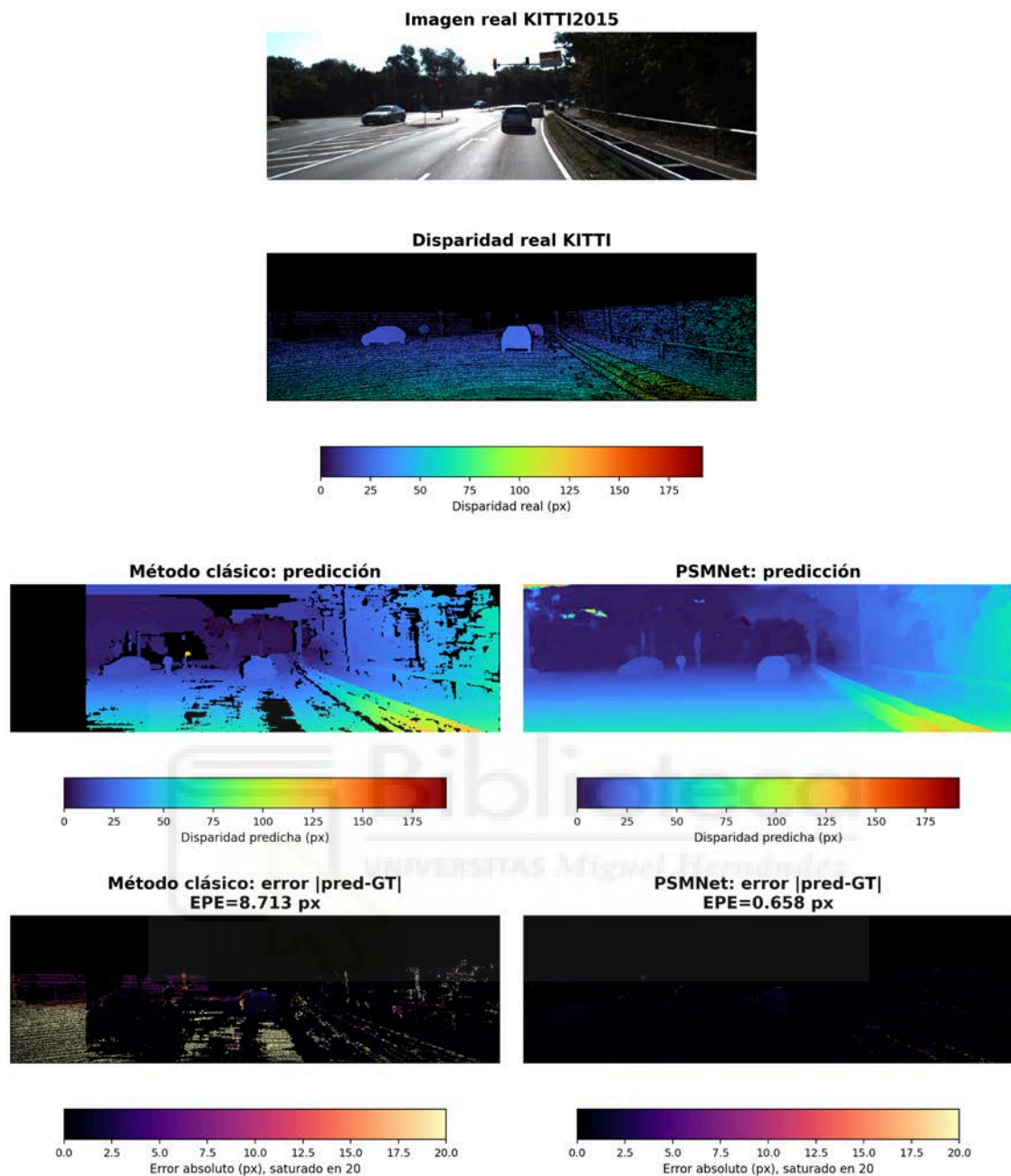


Figura 4.15. Comparación de disparidad y error entre el método clásico y PSMNet en la muestra 000178_10 de KITTI2015.

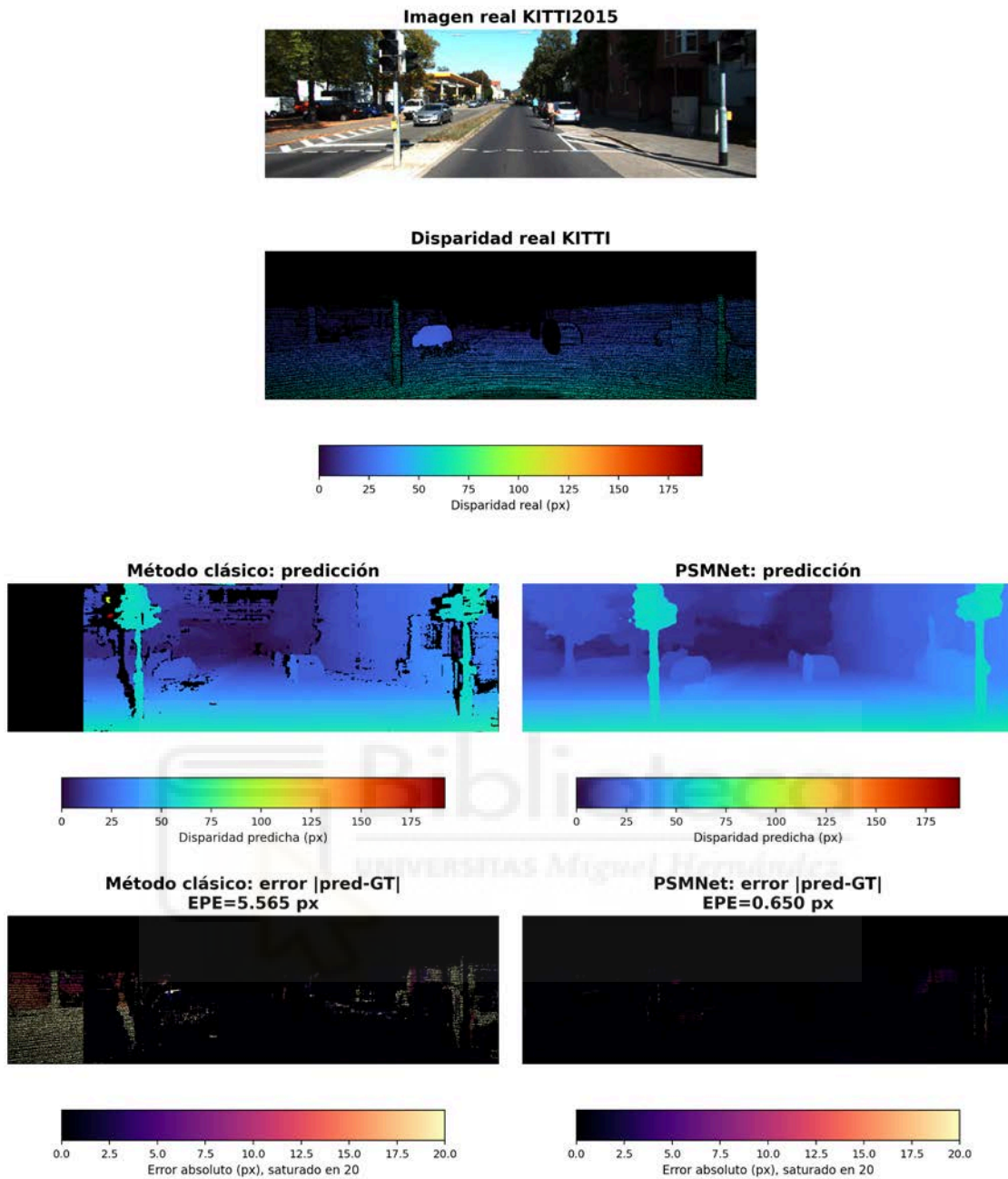


Figura 4.16. Comparación de disparidad y error entre el método clásico y PSMNet en la muestra 000186_10 de KITTI2015.

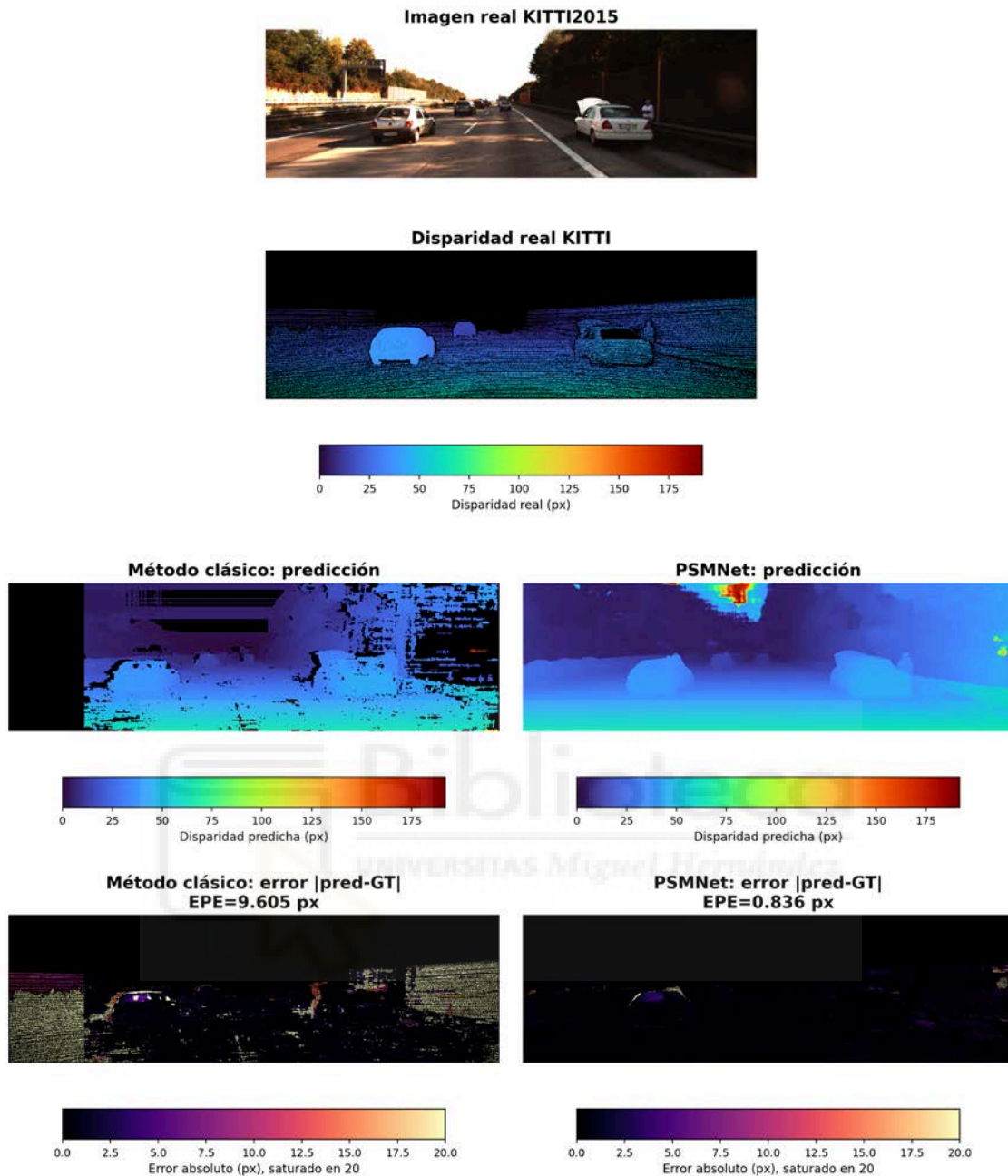


Figura 4.17. Comparación de disparidad y error entre el método clásico y PSMNet en la muestra 000198_10 de KITTI2015.

Se puede observar que la predicción de disparidad, entre el punto de partida con el método geométrico y la basada en PSMNet ajustada, ha mejorado considerablemente.

La predicción de PSMNet en regiones superiores, como el cielo, en algunos casos puede no corresponderse con la imagen real. Por ejemplo, en la *Figura 4.17*, la predicción hecha por la cnn muestra en la zona superior una zona con

alta disparidad, en rojo, correspondiente al cielo. Esto puede deberse a que los mapas de disparidad real no incluyen datos en las regiones superiores, no permitiendo un aprendizaje preciso en el entrenamiento para estas zonas.

4.5. RECONSTRUCCIÓN 3D

4.5.1. DESCRIPCIÓN DEL PROCESO DE RECONSTRUCCIÓN 3D

Como última fase experimental del trabajo, se implementó un código de reconstrucción 3D a partir de la disparidad predicha por PSMNet. A diferencia de los apartados anteriores, en esta parte no se busca calcular nuevas métricas cuantitativas, sino comprobar visualmente si el modelo entrenado permite obtener una representación tridimensional coherente de la escena. El código completo utilizado en esta fase se incluye en el *ANEXO 5*.

Para esta reconstrucción se emplearon pares estéreo del conjunto KITTI2015 test. Estos datos se utilizaron porque permiten aplicar el modelo sobre escenas distintas a las usadas en entrenamiento y evaluación, y porque proporcionan los archivos de calibración necesarios para transformar la disparidad estimada en profundidad métrica. El modelo utilizado fue PSMNet con los pesos seleccionados tras el fine-tuning del experimento de capas medias y finales, correspondientes a la época 263.

El funcionamiento general del código puede resumirse en las siguientes etapas:

1. Preparación del entorno y carga de datos.
En primer lugar, el código monta Google Drive, comprueba la disponibilidad de GPU y carga las rutas necesarias. Después descomprime el conjunto KITTI2015 test y sus archivos de calibración. A partir de las imágenes disponibles, se seleccionan manualmente las muestras sobre las que se desea realizar la reconstrucción.
2. Carga del modelo PSMNet.
Una vez preparado el conjunto de datos, se importa la arquitectura PSMNet desde el repositorio utilizado en el proyecto y se cargan los pesos seleccionados. Antes de realizar la inferencia, el código verifica que el checkpoint contiene las claves esperadas y que su estructura es compatible con el modelo.
3. Lectura de calibración de cada muestra.
Para cada par estéreo se lee el archivo de calibración correspondiente. De este archivo se extraen los parámetros necesarios para la reconstrucción, como la distancia focal, el centro óptico y la baseline

entre cámaras. Estos valores se obtienen a partir de las matrices de proyección asociadas a las cámaras izquierda y derecha. Aunque los parámetros puedan ser similares entre distintas muestras, se lee la calibración de cada imagen para evitar asumir una configuración constante en todo el conjunto.

4. Inferencia de disparidad.

El par estéreo formado por la imagen izquierda y derecha se introduce en PSMNet. Antes de la inferencia, las imágenes se adaptan al tamaño requerido por la red mediante padding. Como salida se obtiene un mapa de disparidad denso, que indica el desplazamiento horizontal estimado entre píxeles correspondientes de ambas imágenes.

5. Estimación de profundidad.

A partir del mapa de disparidad predicho y de los parámetros de calibración se calcula la profundidad de cada píxel válido. En este apartado no se repite la formulación geométrica, ya explicada previamente, pero sí debe destacarse que esta etapa permite pasar de una predicción en píxeles a una estimación en metros. El resultado se guarda tanto como matriz numérica como en forma de imagen 2D titulada “Estimación de profundidad”, con una barra de color que representa la profundidad en metros.

6. Generación de la nube de puntos 3D.

Una vez calculada la profundidad, cada píxel válido de la imagen se re proyecta al espacio tridimensional. Para ello se utilizan las coordenadas del píxel, la profundidad estimada y los parámetros internos de la cámara. El color de cada punto se toma de la imagen izquierda original, de forma que la nube conserva la apariencia visual de la escena. Además, se filtran puntos inválidos o fuera del rango de profundidad establecido y se aplica un muestreo espacial para reducir el número de puntos y facilitar la visualización.

7. Guardado y visualización de resultados.

Para cada muestra se guardan distintos archivos de salida. En primer lugar, se almacena la disparidad predicha y una visualización titulada “Predicción de disparidad”, con escala en píxeles. También se guarda la estimación de profundidad en metros, la nube de puntos en formato PLY y una versión interactiva en HTML. Esta última permite rotar, ampliar y explorar la nube de puntos desde distintas perspectivas. Además, se genera una vista estática RGB de la reconstrucción 3D y una imagen resumen vertical que combina, de arriba abajo, la imagen izquierda original, la predicción de disparidad, la estimación de profundidad y la vista 3D estática.

4.5.2. RESULTADOS VISUALES DE LA RECONSTRUCCIÓN 3D

La imagen resumen facilita la interpretación de los resultados, ya que permite observar en una única figura la relación entre la escena original, la disparidad calculada por la red, la profundidad estimada y la reconstrucción tridimensional resultante. En la memoria se incluyen estas imágenes resumen para las muestras seleccionadas, junto con capturas adicionales de la nube de puntos interactiva desde diferentes perspectivas.

El principal añadido de este código respecto a las etapas anteriores es la transformación de la disparidad en una representación 3D. En las fases de evaluación se analizaba principalmente el error de disparidad, mientras que en esta etapa se utiliza dicha disparidad para obtener profundidad y coordenadas espaciales. Por tanto, el resultado final ya no es únicamente un mapa 2D, sino una nube de puntos que representa la geometría aproximada de la escena.

Debe tenerse en cuenta que la calidad de la reconstrucción depende directamente de la calidad de la disparidad estimada. Si la red comete errores en zonas con poca textura, reflejos, cielo, objetos lejanos o bordes finos, esos errores se trasladan también a la profundidad y a la nube de puntos. Por este motivo, algunas zonas pueden aparecer incompletas, ruidosas o con puntos mal localizados. Aun así, los resultados permiten comprobar que el modelo es capaz de recuperar la estructura general de la escena y generar una reconstrucción visualmente interpretable.

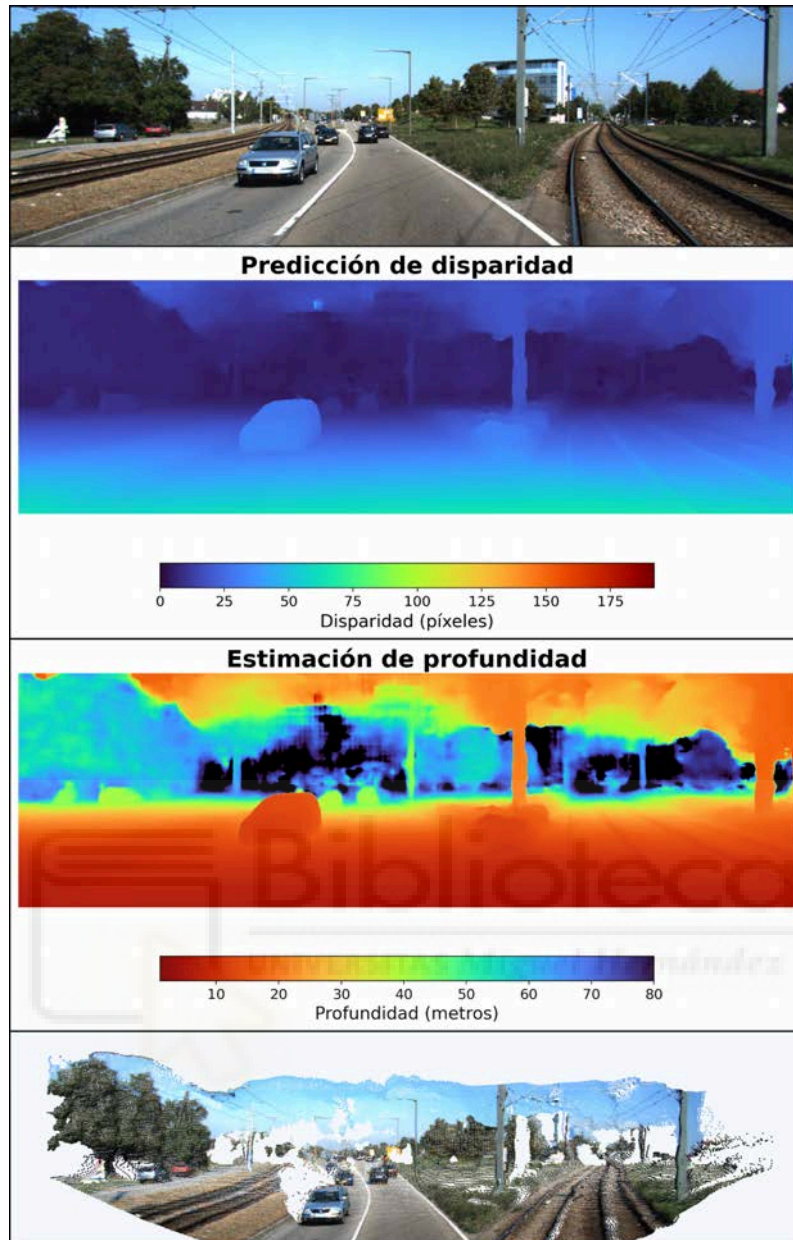


Figura 4.18. Comparativa de resultados finales, primer ejemplo.

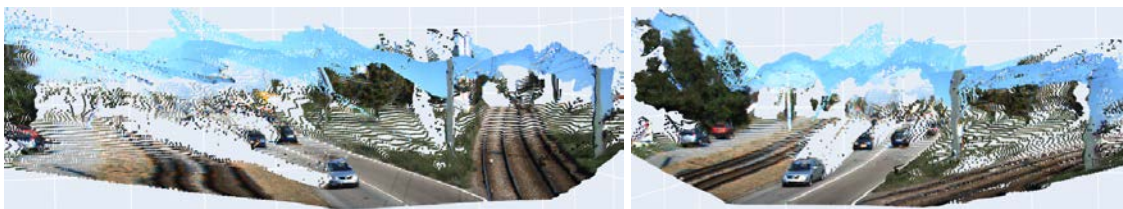


Figura 4.19. Diferentes perspectivas de la nube de puntos, primer ejemplo.

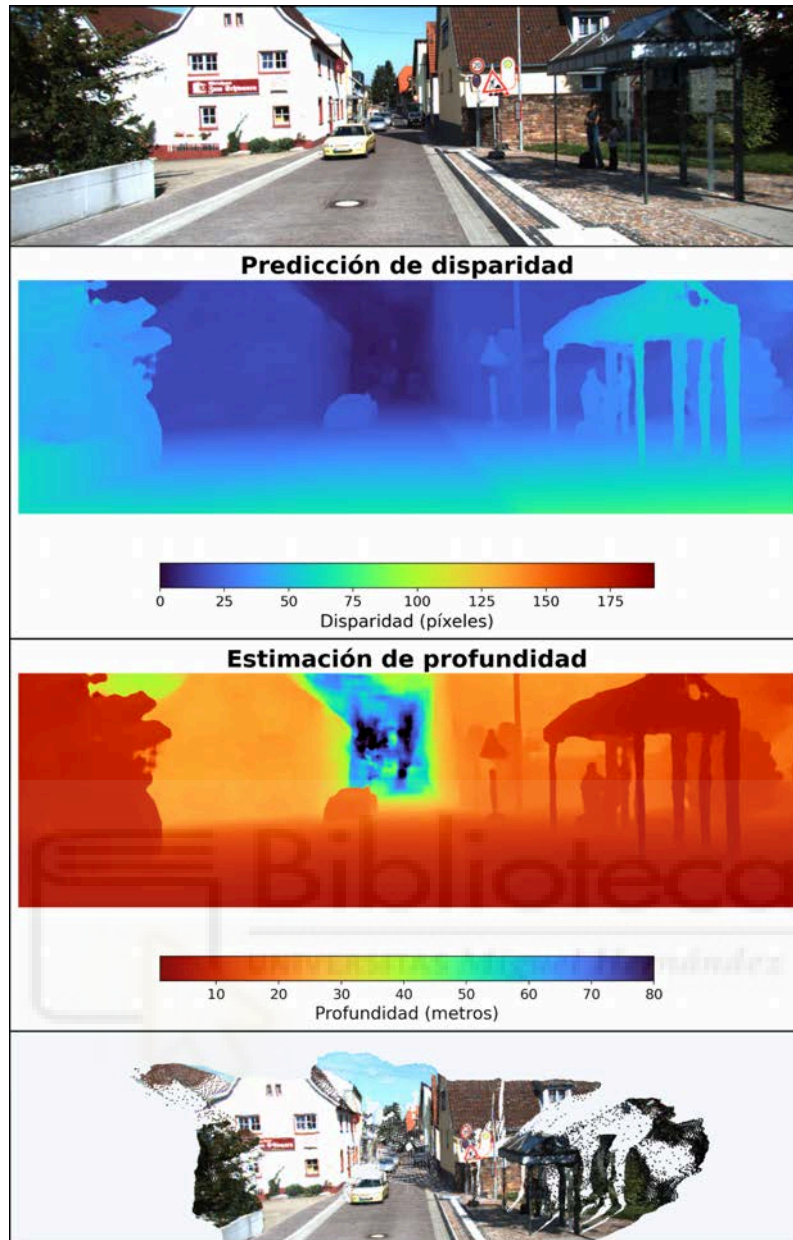


Figura 4.20. Comparativa de resultados finales, segundo ejemplo.



Figura 4.21. Diferentes perspectivas de la nube de puntos, segundo ejemplo.

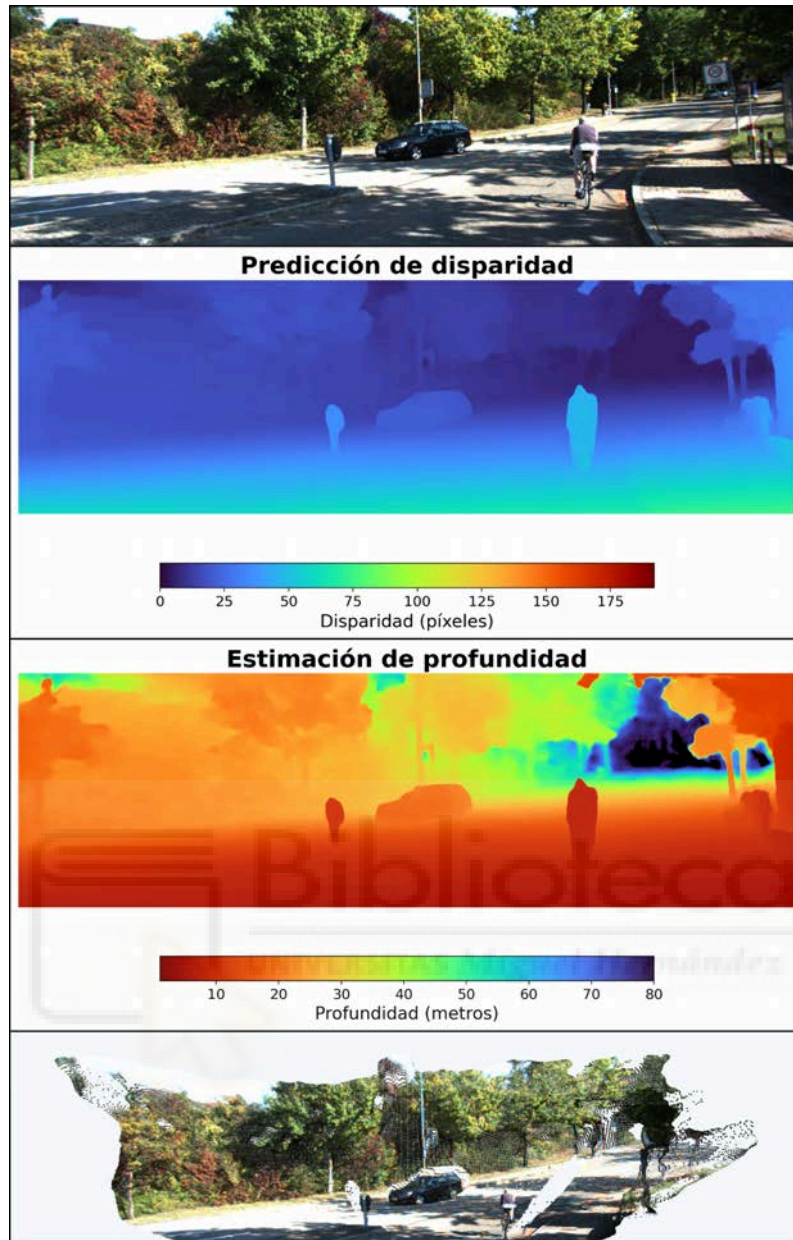


Figura 4.22. Comparativa de resultados finales, tercer ejemplo.



Figura 4.23. Diferentes perspectivas de la nube de puntos, tercer ejemplo.

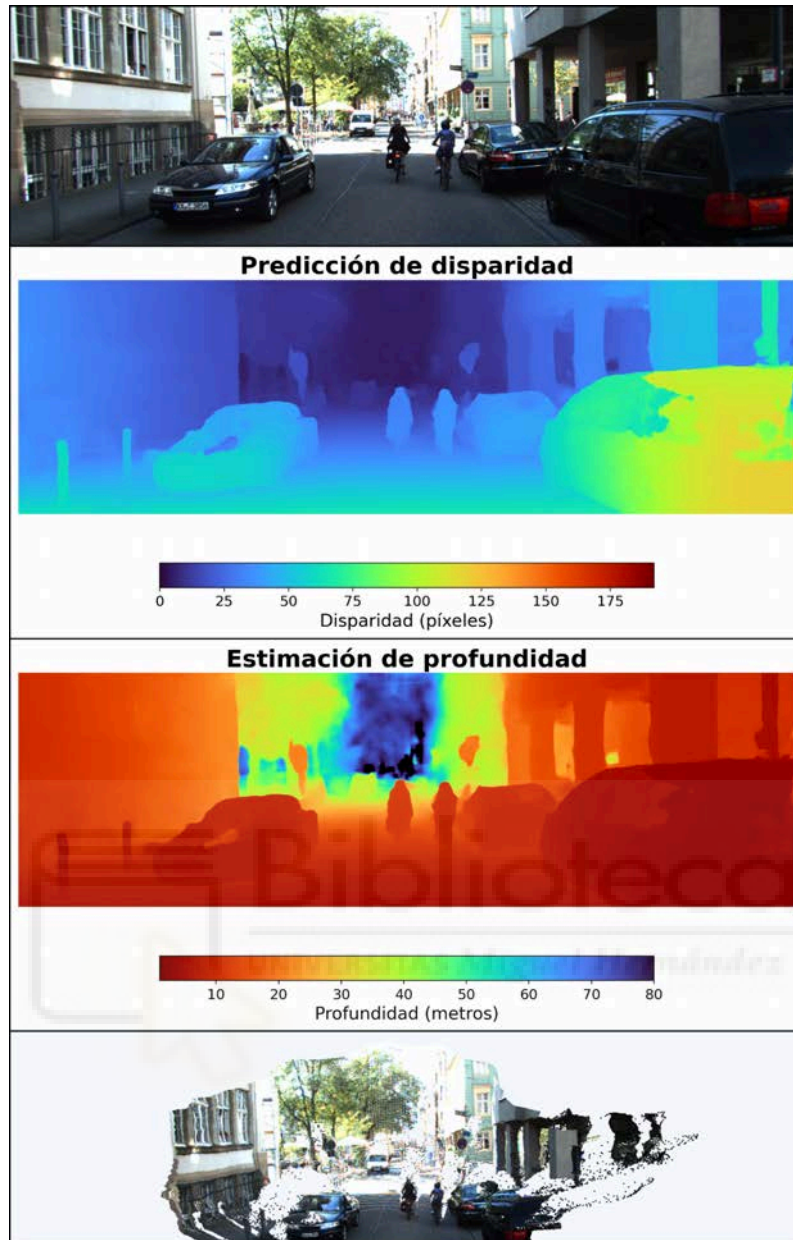


Figura 4.24. Comparativa de resultados finales, cuarto ejemplo.



Figura 4.25. Diferentes perspectivas de la nube de puntos, cuarto ejemplo.

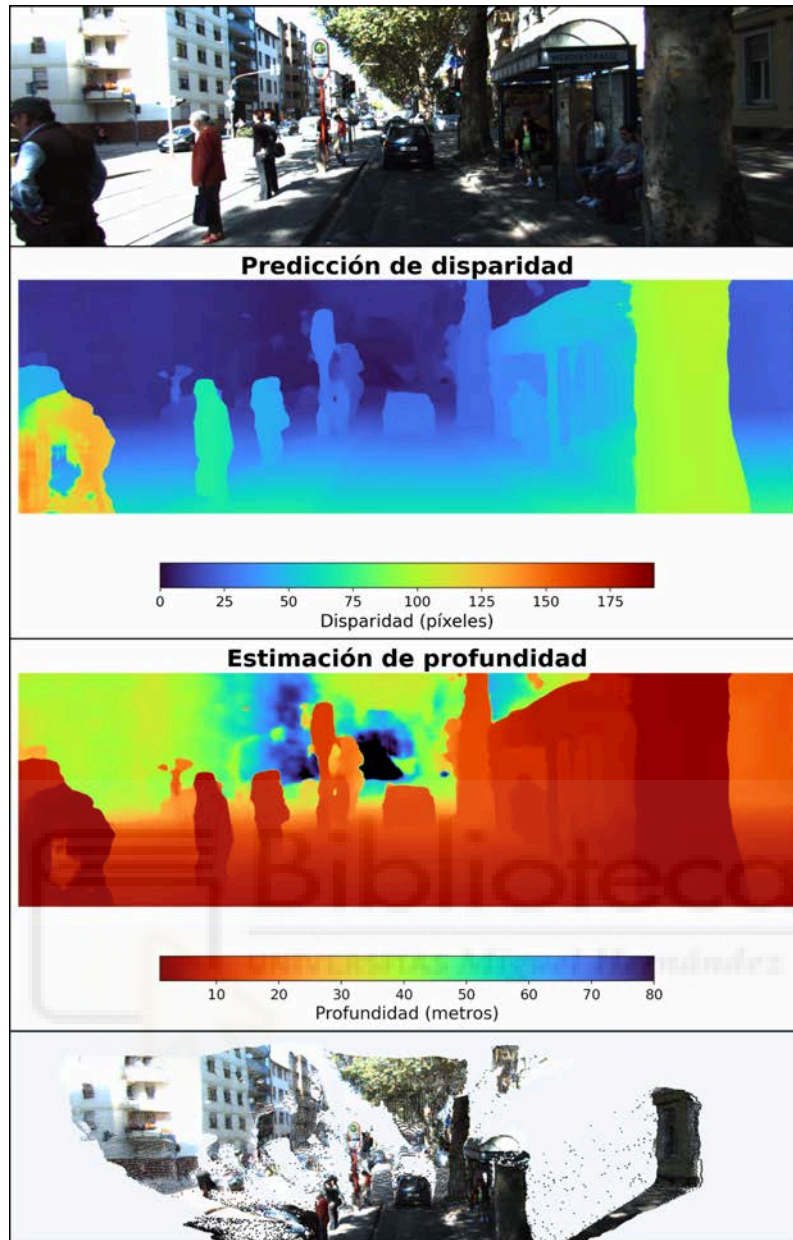


Figura 4.26. Comparativa de resultados finales, quinto ejemplo.



Figura 4.27. Diferentes perspectivas de la nube de puntos, quinto ejemplo.

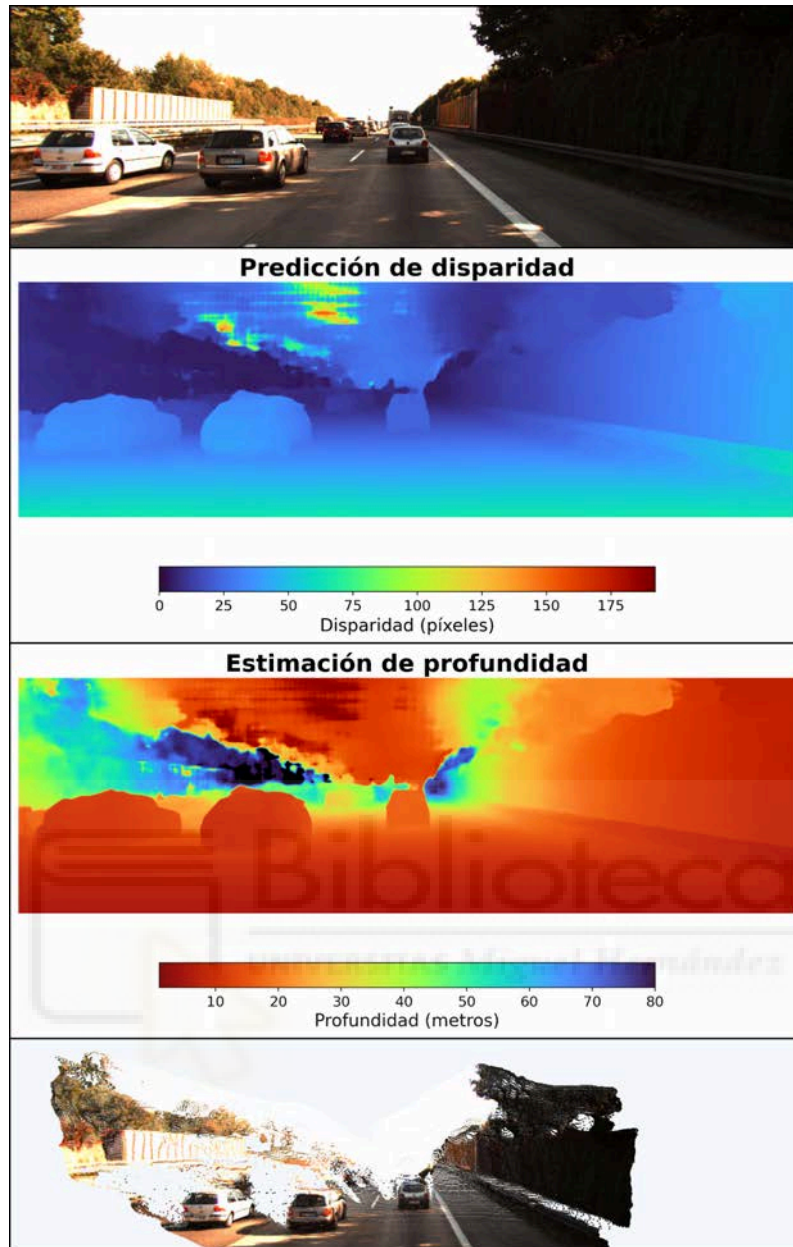


Figura 4.28. Comparativa de resultados finales, sexto ejemplo.

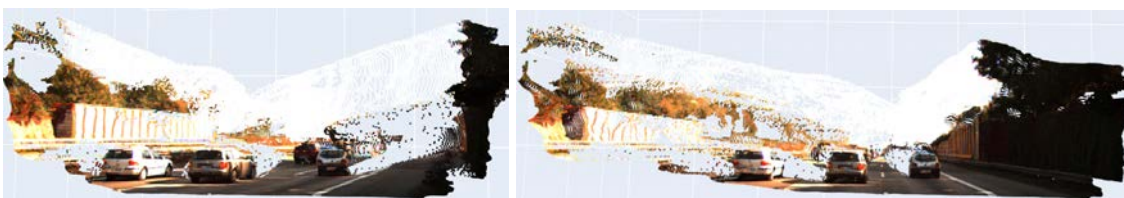


Figura 4.29. Diferentes perspectivas de la nube de puntos, sexto ejemplo.

En los resultados anteriores se observa que, en algunas muestras, aparecen puntos 3D en zonas que físicamente no deberían formar parte de la nube de puntos reconstruida. Un ejemplo claro se encuentra en la *Figura 4.19* y en la *Figura 4.29*, donde se aprecia la presencia del cielo dentro de la nube de puntos. Desde el punto de vista geométrico, esta región debería tender a una profundidad muy elevada o incluso al infinito, por lo que no debería representarse como una superficie dentro del rango de reconstrucción considerado.

Este efecto se debe a que PSMNet genera una predicción densa de disparidad para la imagen completa, incluyendo zonas en las que la correspondencia estéreo es muy ambigua. El cielo suele presentar poca textura, variaciones suaves de color y ausencia de elementos claramente emparejables entre la imagen izquierda y derecha. Por ello, la red puede asignar valores de disparidad no nulos a estas regiones, aunque en realidad no exista una superficie cercana medible. Al transformar esa disparidad predicha en profundidad, el código interpreta esos valores como puntos situados a una distancia finita. Si dicha profundidad queda dentro del rango válido establecido en el código, entre 1 m y 80 m, los puntos se incorporan a la nube 3D.

Además, esta situación está relacionada con la forma en que KITTI proporciona la disparidad real. En las zonas superiores de la imagen, especialmente en el cielo, normalmente no existe disparidad real válida, ya que las etiquetas de referencia se obtienen a partir de sensores LiDAR proyectados sobre la imagen. El LiDAR no proporciona medidas útiles del cielo y, por tanto, esas regiones aparecen como no válidas en los mapas de disparidad real. Durante la evaluación, estos píxeles se excluyen mediante la máscara de validez, por lo que los errores en el cielo no se penalizan del mismo modo que en zonas con ground truth disponible.

Por tanto, la presencia del cielo en la nube de puntos no se debe a un fallo en la fórmula de reconstrucción 3D, sino a una predicción de disparidad físicamente incorrecta en una región sin información estéreo fiable. El código elimina profundidades infinitas, no finitas o superiores al límite establecido, pero no puede saber por sí mismo que una zona pertenece al cielo si la red le asigna una disparidad válida. Para evitar este efecto podrían aplicarse filtros adicionales, como eliminar una franja superior de la imagen, limitar más estrictamente la profundidad máxima o incorporar una segmentación semántica que descarte regiones clasificadas como cielo.

También se observan zonas incompletas en la nube de puntos debidas a oclusiones propias de la visión estéreo. Este efecto es especialmente visible en la *Figura 4.27*, donde algunos objetos o regiones quedan parcialmente reconstruidos o presentan huecos. Esto era esperable, ya que existen zonas visibles desde una cámara que no aparecen de la misma forma en la otra, por lo que la correspondencia estéreo resulta ambigua o directamente imposible. En consecuencia, la disparidad estimada en esas regiones puede ser incorrecta o no suficientemente fiable, generando pérdidas de información en la reconstrucción 3D.

4.5.3. CALIBRACIÓN Y RECTIFICACIÓN EPIPOLAR

Respecto a la calibración y rectificación epipolar, en el código final no se aplica una rectificación adicional sobre las imágenes de KITTI. Esta decisión se debe a que KITTI proporciona pares estéreo previamente rectificados. PSMNet asume que las correspondencias entre ambas imágenes se encuentran en la misma línea horizontal, y esta condición ya se cumple en las imágenes empleadas. Por ello, el código utiliza directamente los pares estéreo proporcionados por KITTI y lee los parámetros de calibración asociados a cada muestra para realizar la reconstrucción métrica.

Como comprobación adicional, se realizó una prueba experimental aplicando una nueva rectificación epipolar sobre imágenes de KITTI ya rectificadas. Para ello se emplearon funciones de OpenCV como `stereoRectify`, `initUndistortRectifyMap` y `remap`, utilizando los parámetros de calibración proporcionados por KITTI. El procedimiento consistió en calcular nuevas matrices de rectificación, generar los mapas de remapeo de ambas cámaras y transformar las imágenes izquierda y derecha antes de introducirlas en PSMNet.

El resultado de esta prueba, un ejemplo es la *Figura 4.30*, mostró una degradación clara de la predicción de disparidad, de la estimación de profundidad y de la nube de puntos 3D. Esta degradación se debe a que volver a rectificar imágenes que ya han sido rectificadas modifica innecesariamente la entrada del modelo. Aunque la rectificación epipolar busca alinear las correspondencias en la misma coordenada vertical, el remapeo puede introducir interpolaciones, cambios de escala, desplazamientos, recortes o deformaciones. Estas modificaciones alteran el par estéreo original y afectan directamente a la predicción de la red. Por esta razón, la rectificación adicional no se incluye en el código final, y la prueba se mantiene únicamente como justificación visual de esta decisión metodológica.

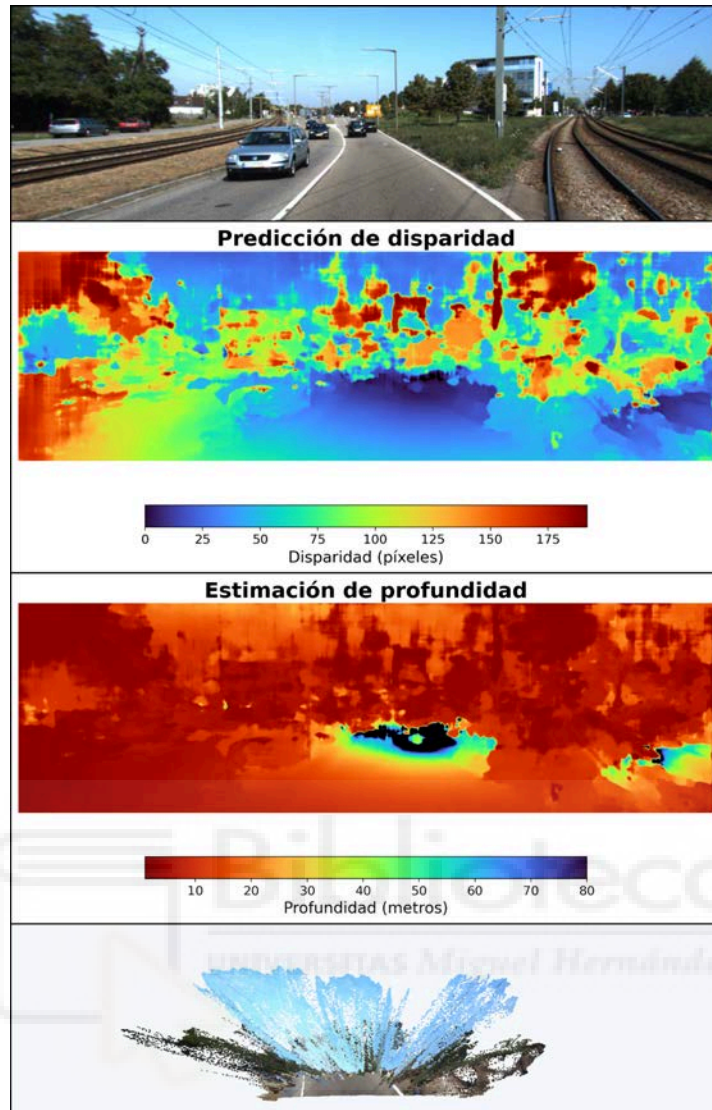


Figura 4.30. Resultado de aplicar rectificación epipolar adicional sobre imágenes KITTI ya rectificadas.

En un sistema general con cámaras propias, el procedimiento sería diferente. En ese caso sí sería necesario realizar una calibración intrínseca de cada cámara para obtener sus parámetros internos y la distorsión de las lentes. Posteriormente, habría que realizar una calibración estéreo para calcular la relación geométrica entre ambas cámaras, incluyendo la rotación, la traslación y la baseline. Con estos datos se generarían los mapas de rectificación epipolar, que se aplicarían a cada par de imágenes antes de introducirlas en PSMNet. Esta etapa se realizaría normalmente en un código independiente de calibración, ejecutado una vez mientras no cambie el montaje físico de las cámaras. En este trabajo no ha sido necesario porque KITTI ya proporciona imágenes rectificadas y calibradas.

5. CONCLUSIONES

Este Trabajo Fin de Grado ha permitido desarrollar un flujo completo de reconstrucción 3D a partir de imágenes estéreo, desde la estimación de disparidad hasta la generación de nubes de puntos. La principal conclusión general es que la calidad de la reconstrucción no depende únicamente de elegir una arquitectura de red neuronal, sino de integrar correctamente todas las etapas del proceso: preparación de datos, calibración, entrenamiento, selección de pesos, evaluación y conversión geométrica de disparidad a profundidad.

5.1. CONCLUSIONES PRINCIPALES DEL TRABAJO

El desarrollo del proyecto confirma que las redes neuronales convolucionales son una herramienta adecuada para abordar el problema de la estimación de disparidad en visión estéreo. En particular, PSMNet ha demostrado ser una arquitectura capaz de generar mapas de disparidad densos y coherentes a partir de pares de imágenes rectificadas, lo que permite posteriormente obtener información de profundidad y reconstrucciones tridimensionales del entorno. Una conclusión importante es que el aprendizaje profundo no sustituye por completo a la geometría clásica, sino que la complementa. La red neuronal se encarga de estimar la disparidad, pero la obtención de profundidad métrica sigue dependiendo de los parámetros físicos del sistema estéreo, como la distancia focal y la línea base entre cámaras. Por tanto, incluso utilizando una CNN, conceptos como calibración, rectificación, disparidad y triangulación siguen siendo fundamentales para que la reconstrucción 3D tenga sentido físico.

También se ha comprobado que la transferencia de aprendizaje resulta especialmente útil en este tipo de problemas. Partir de un modelo preentrenado permite aprovechar conocimiento aprendido previamente y adaptarlo al dominio de trabajo mediante fine-tuning. Además, los experimentos realizados muestran que no siempre es necesario reentrenar toda la red; en algunos casos, ajustar solo determinadas capas puede ser una estrategia suficiente y más eficiente. Esto es relevante porque reduce el coste computacional y permite adaptar el modelo sin modificar por completo las representaciones aprendidas durante el preentrenamiento.

Otra conclusión destacable es la importancia de separar correctamente las fases de entrenamiento, selección de modelo y evaluación final. Utilizar datos

distintos para cada etapa permite obtener una valoración más fiable del comportamiento del modelo y evita que las conclusiones se basen en resultados demasiado optimistas. Del mismo modo, seleccionar los pesos según una métrica de validación resulta más adecuado que emplear directamente la última época entrenada, ya que un mayor número de épocas no garantiza necesariamente un mejor resultado final.

En conjunto, el trabajo demuestra que es posible construir un sistema funcional de reconstrucción 3D basado en CNN a partir de imágenes estéreo. No obstante, también queda claro que el resultado final depende tanto del modelo como de las decisiones metodológicas tomadas durante todo el proceso.

5.2. ANÁLISIS DE RESULTADOS Y LIMITACIONES

Los resultados obtenidos muestran que PSMNet ofrece mapas de disparidad más estables y coherentes que el método clásico evaluado. Esta diferencia se aprecia especialmente en zonas donde la correspondencia estéreo resulta más difícil, como regiones con poca textura, superficies uniformes, bordes de objetos, oclusiones o cambios de iluminación. Mientras que el método clásico depende en gran medida de la comparación local entre píxeles, la red neuronal puede aprovechar información contextual aprendida durante el entrenamiento, lo que mejora la consistencia global de la estimación.

La comparación con el método clásico también permite observar un compromiso claro entre calidad y coste computacional. El método clásico presenta la ventaja de ser más rápido y sencillo, además de no requerir entrenamiento previo ni uso intensivo de GPU. Sin embargo, sus mapas de disparidad son más sensibles al ruido y a las ambigüedades de la escena. PSMNet, por el contrario, consigue mejores resultados en términos de calidad de disparidad y reconstrucción, aunque a costa de una mayor carga computacional.

La reconstrucción 3D final confirma que los mapas de disparidad estimados por la red pueden transformarse en nubes de puntos interpretables. A partir de la disparidad y de los parámetros de calibración, se obtiene una representación tridimensional que permite visualizar la estructura espacial de la escena. Esto demuestra la utilidad de la visión estéreo como técnica pasiva para obtener información de profundidad sin recurrir a sensores activos como LiDAR.

Sin embargo, el trabajo también presenta limitaciones. La primera es la dependencia de la calibración: si los parámetros de cámara no son correctos, la profundidad calculada puede verse afectada aunque el mapa de disparidad sea razonable. Otra limitación es que la evaluación se ha realizado principalmente sobre KITTI, por lo que los resultados no pueden generalizarse directamente a otros entornos, cámaras o condiciones de iluminación sin realizar nuevas pruebas.

Además, la reconstrucción generada a partir de una única pareja estéreo proporciona una nube de puntos parcial de la escena, no una reconstrucción completa del entorno. Esto hace que las oclusiones sean especialmente visibles en la nube de puntos, ya que las zonas que no son observables desde el par estéreo no pueden reconstruirse correctamente y aparecen como huecos, discontinuidades o sombras geométricas. También se han observado puntos generados en regiones correspondientes al cielo, donde la estimación de disparidad no siempre es fiable por la ausencia de textura y de una geometría definida.

También hay que tener en cuenta que los errores de disparidad se amplifican al convertirlos en profundidad, especialmente en zonas alejadas, donde pequeñas variaciones de disparidad pueden producir diferencias importantes en distancia. Por último, aunque PSMNet mejora la calidad frente al método clásico, su coste computacional limita su aplicación directa en tiempo real si no se aplican técnicas de optimización.

5.3. TRABAJOS FUTUROS

Como continuación de este trabajo, una primera línea futura sería evaluar el sistema en otros conjuntos de datos y en imágenes capturadas con una cámara estéreo propia. Esto permitiría analizar la capacidad de generalización del modelo fuera del entorno KITTI y comprobar su comportamiento ante distintas condiciones de iluminación, texturas, resoluciones y configuraciones de cámara.

Otra mejora posible sería comparar PSMNet con arquitecturas de estimación estéreo más recientes. Existen modelos posteriores que buscan mejorar la precisión, reducir el tiempo de inferencia o aumentar la robustez frente a cambios de dominio. Evaluarlos bajo la misma metodología permitiría determinar si ofrecen ventajas significativas para la reconstrucción 3D.

También sería interesante ampliar el estudio del fine-tuning. Se podrían probar nuevas combinaciones de capas entrenables, diferentes tasas de aprendizaje, técnicas de aumento de datos o estrategias de parada temprana. Esto permitiría encontrar un mejor equilibrio entre precisión, tiempo de entrenamiento y capacidad de generalización.

En cuanto a la reconstrucción 3D, se podrían aplicar técnicas de postprocesado sobre las nubes de puntos, como eliminación de outliers, filtrado de puntos erróneos, segmentación del suelo o fusión de varias reconstrucciones. En particular, para reducir los puntos generados en la zona del cielo, se podrían estudiar filtros basados en altura, profundidad y conectividad, eliminando regiones de puntos aislados o poco consistentes que no formen parte de estructuras reales de la escena. Para mitigar los huecos producidos por las oclusiones, una extensión natural sería trabajar con secuencias de imágenes en lugar de pares aislados, acumulando nubes de puntos de diferentes instantes de una misma escena. Para ello sería necesario estimar o conocer el movimiento de la cámara y alinear las reconstrucciones parciales en un mismo sistema de referencia, de forma que distintas vistas puedan complementar zonas que en una única pareja estéreo quedan ocultas.

Finalmente, una línea importante de mejora sería optimizar el tiempo de inferencia del modelo. Para ello podrían estudiarse técnicas como precisión mixta, reducción del tamaño de la red, exportación a formatos optimizados o uso de arquitecturas más ligeras. Esto permitiría avanzar hacia un sistema de reconstrucción 3D estéreo más eficiente y aplicable en escenarios reales.

6. BIBLIOGRAFÍA

- [1] R. Hartley y A. Zisserman, Multiple View Geometry in Computer Vision, 2.^a ed. Cambridge: Cambridge University Press, 2004.
- [2] R. Szeliski, Computer Vision: Algorithms and Applications, 2.^a ed. Cham: Springer, 2022.
- [3] H. Hirschmüller, “Stereo Processing by Semi-Global Matching and Mutual Information”, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 30, n.º 2, pp. 328-341, 2008. doi: 10.1109/TPAMI.2007.1166.
- [4] OpenCV, “cv::StereoSGBM Class Reference”, OpenCV 4.x documentation. https://docs.opencv.org/4.x/d2/d85/classcv_1_1StereoSGBM.html
- [5] J. Žbontar y Y. LeCun, “Stereo Matching by Training a Convolutional Neural Network to Compare Image Patches”, Journal of Machine Learning Research, vol. 17, n.º 65, pp. 1-32, 2016.
- [6] A. Kendall, H. Martirosyan, S. Dasgupta, P. Henry, R. Kennedy, A. Bachrach y A. Bry, “End-to-End Learning of Geometry and Context for Deep Stereo Regression”, Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2017, pp. 66-75.
- [7] J.-R. Chang y Y.-S. Chen, “Pyramid Stereo Matching Network”, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 5410-5418.
https://openaccess.thecvf.com/content_cvpr_2018/papers/Chang_Pyramid_Stereo_Matching_CVPR_2018_paper.pdf
- [8] JiaRenChang, “PSMNet: Pyramid Stereo Matching Network”, GitHub repository. <https://github.com/JiaRenChang/PSMNet>
- [9] N. Mayer, E. Ilg, P. Häusser, P. Fischer, D. Cremers, A. Dosovitskiy y T. Brox, “A Large Dataset to Train Convolutional Networks for Disparity, Optical Flow, and Scene Flow Estimation”, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.
- [10] The KITTI Vision Benchmark Suite. A project of Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago. <https://cvlibs.net/datasets/kitti/>

7. ANEXOS

7.1. ANEXO 1: EVALUACIÓN DEL MÉTODO CLÁSICO

```
# =====  
# EVALUACIÓN MÉTODO CLÁSICO - KITTI2015 SEGUNDA MITAD  
# Método:  
#   - OpenCV StereoSGBM  
#  
# Evalúa:  
#   - Segunda mitad de KITTI2015 training:  
#       índices 100-199  
#       imágenes _10 y _11  
#  
# Guarda en Drive:  
#   /content/drive/MyDrive/TFG/Evaluacion/MetodoClasico/Resultados/  
#  
# Métricas calculadas:  
#   - D1-all  
#   - EPE  
#   - AbsRel  
#   - delta1  
# =====  
  
import os  
import csv  
import json  
import time  
import shutil  
import zipfile  
from pathlib import Path  
  
import cv2  
import numpy as np  
import torch  
  
from google.colab import drive
```

```

# =====
# 1. CONFIGURACIÓN
# =====

drive.mount("/content/drive", force_remount=True)

CONFIG = {
    "kitti2015_zip_drive":
        "/content/drive/MyDrive/TFG/Datasets/KITTI2015/data_scene_flow.zip",

    "maxdisp": 192,

    "output_dir":
        "/content/drive/MyDrive/TFG/Evaluacion/MetodoClasico/Resultados"
    ,

    # Parámetros StereoSGBM
    "sgbm_min_disparity": 0,
    "sgbm_num_disparities": 192, # debe ser múltiplo de 16 y
    coincidir con maxdisp
    "sgbm_block_size": 5,

    "sgbm_uniqueness_ratio": 10,
    "sgbm_speckle_window_size": 100,
    "sgbm_speckle_range": 2,
    "sgbm_disp12_max_diff": 1,
    "sgbm_pre_filter_cap": 63,

    # Guardado de mapas de disparidad
    "save_disparity_maps": True,

    # Guarda la predicción real en píxeles como .npy
    "save_pred_npy": True,

    # Guarda la predicción como PNG uint16 estilo KITTI:
    # valor_guardado = disparidad * 256
    "save_pred_kitti_png": True,

```

```

    # Guarda imágenes visuales normalizadas solo para inspección
visual
    "save_visual_maps": True,

    # Guarda visualizaciones del GT y del error absoluto
    "save_gt_visual_maps": True,
    "save_error_visual_maps": True,
    "max_visual_maps_to_save": None,

    # Rango máximo del error absoluto para visualización.
    # Los errores mayores se saturan a este valor.
    "error_visual_max_px": 20.0,

    # Para desconectar Colab al final
    "auto_disconnect": False,
}

# =====
# 2. UTILIDADES GENERALES
# =====

def ensure_dir(path):
    Path(path).mkdir(parents=True, exist_ok=True)

def remove_dir_if_exists(path):
    p = Path(path)
    if p.exists():
        shutil.rmtree(p)

def copy_zip_from_drive_to_content(src_drive_path,
dst_dir="/content/datasets_zip"):
    ensure_dir(dst_dir)

    src = Path(src_drive_path)

    if not src.exists():

```

```

        raise FileNotFoundError(f"No existe el ZIP en Drive:
{src_drive_path}")

    dst = Path(dst_dir) / src.name

    print("Copiando ZIP desde Drive:")
    print(f" Origen: {src}")
    print(f" Destino: {dst}")

    shutil.copy2(src, dst)

    if not dst.exists() or dst.stat().st_size == 0:
        raise RuntimeError(f"Error al copiar ZIP: {dst}")

    return str(dst)

def unzip_file(zip_path, extract_to, clean_before_extract=True):
    if clean_before_extract:
        remove_dir_if_exists(extract_to)

    ensure_dir(extract_to)

    print("Descomprimiendo:")
    print(f" ZIP: {zip_path}")
    print(f" Destino: {extract_to}")

    try:
        with zipfile.ZipFile(zip_path, "r") as zf:
            bad_member = zf.testzip()

            if bad_member is not None:
                raise RuntimeError(
                    f"ZIP corrupto o incompleto. Primer archivo
problemático: {bad_member}"
                )

            zf.extractall(extract_to)

    except zipfile.BadZipFile:

```

```

        raise RuntimeError(f"Archivo ZIP no válido: {zip_path}")

    return extract_to

def find_kitti2015_training_root(search_root):
    required = {"image_2", "image_3", "disp_occ_0"}

    for root, dirs, files in os.walk(search_root):
        if required.issubset(set(dirs)):
            return root + "/"

    raise FileNotFoundError(
        f"No se encontró la raíz training de KITTI2015 dentro de {search_root}"
    )

def prepare_kitti2015(config):
    local_zip_dir = "/content/datasets_zip"

    kitti_zip = copy_zip_from_drive_to_content(
        config["kitti2015_zip_drive"],
        local_zip_dir
    )

    kitti_extract = unzip_file(
        kitti_zip,
        "/content/datasets/kitti2015",
        clean_before_extract=True
    )

    kitti_root = find_kitti2015_training_root(kitti_extract)

    print("KITTI2015 preparado correctamente.")
    print(f"Raíz imágenes/GT: {kitti_root}")

    return kitti_root

```

```

# =====
# 3. LISTAS KITTI2015 SEGUNDA MITAD
# =====

def build_kitti2015_final_eval_lists_second_half(kitti2015_root):
    left_dir = Path(kitti2015_root) / "image_2"
    right_dir = Path(kitti2015_root) / "image_3"
    disp_dir = Path(kitti2015_root) / "disp_occ_0"

    requested = []

    for i in range(100, 200):
        requested.append(f"{i:06d}_10.png")
        requested.append(f"{i:06d}_11.png")

    left_names = {p.name for p in left_dir.glob("*.png")}
    right_names = {p.name for p in right_dir.glob("*.png")}
    disp_names = {p.name for p in disp_dir.glob("*.png")}

    common = []
    missing = []

    for n in requested:
        ok = (
            n in left_names
            and n in right_names
            and n in disp_names
        )

        if ok:
            common.append(n)
        else:
            missing.append(n)

    if len(common) == 0:
        raise RuntimeError(
            "No se encontraron muestras válidas en la segunda mitad
de KITTI2015."
        )

```



```

    if len(missing) > 0:
        print(f"[WARN] Hay {len(missing)} muestras solicitadas que
no se encontraron completas.")
        print("Primeras muestras omitidas:", missing[:10])

    left = [str(left_dir / n) for n in common]
    right = [str(right_dir / n) for n in common]
    disp = [str(disp_dir / n) for n in common]

    print(f"KITTI2015 evaluación método clásico: {len(common)}
muestras válidas.")

    return left, right, disp, common

# =====
# 4. LECTURA DE DISPARIDAD REAL KITTI
# =====

def read_kitti_disparity_png(path):
    """
    KITTI guarda la disparidad real como PNG uint16 escalado por
    256.
    La disparidad real en píxeles es:
        disp = valor_png / 256.0
    """

    disp_raw = cv2.imread(path, cv2.IMREAD_UNCHANGED)

    if disp_raw is None:
        raise FileNotFoundError(f"No se pudo leer la disparidad GT:
{path}")

    disp = disp_raw.astype(np.float32) / 256.0

    return disp

```

```

# =====
# 5. MÉTODO CLÁSICO: StereoSGBM
# =====

def create_sgbm_matcher(config):
    num_disparities = config["sgbm_num_disparities"]

    if num_disparities % 16 != 0:
        raise ValueError("sgbm_num_disparities debe ser múltiplo de
16.")

    if config["sgbm_min_disparity"] != 0:
        raise ValueError(
            "Para comparar limpiamente con PSMNet, se espera
sgbm_min_disparity = 0."
        )

    if config["sgbm_num_disparities"] != config["maxdisp"]:
        raise ValueError(
            "Para una comparación limpia, sgbm_num_disparities debe
coincidir con maxdisp."
        )

    block_size = config["sgbm_block_size"]

    if block_size % 2 == 0 or block_size < 3:
        raise ValueError("sgbm_block_size debe ser impar y >= 3.")

    # En escala de grises, número de canales = 1
    channels = 1

    P1 = 8 * channels * block_size ** 2
    P2 = 32 * channels * block_size ** 2

    matcher = cv2.StereoSGBM_create(
        minDisparity=config["sgbm_min_disparity"],
        numDisparities=num_disparities,
        blockSize=block_size,
        P1=P1,
        P2=P2,

```

```

        disp12MaxDiff=config["sgbm_disp12_max_diff"],
        preFilterCap=config["sgbm_pre_filter_cap"],
        uniquenessRatio=config["sgbm_uniqueness_ratio"],
        speckleWindowSize=config["sgbm_speckle_window_size"],
        speckleRange=config["sgbm_speckle_range"],
        mode=cv2.STEREO_SGBM_MODE_SGBM_3WAY,
    )

    return matcher

def compute_classic_disparity_sgbm(left_path, right_path, matcher):
    """
    Calcula la disparidad con StereoSGBM.

    Criterio temporal análogo al usado en PSMNet:
    - PSMNet mide solo model(imgL, imgR)
    - Método clásico mide solo matcher.compute(imgL, imgR)

    No incluye:
    - lectura de imágenes desde disco
    - conversión de la salida fixed-point de OpenCV a píxeles
    reales
    - lectura de GT
    - cálculo de métricas
    - guardado de resultados
    - creación del matcher
    """

    imgL = cv2.imread(left_path, cv2.IMREAD_GRAYSCALE)
    imgR = cv2.imread(right_path, cv2.IMREAD_GRAYSCALE)

    if imgL is None:
        raise FileNotFoundError(f"No se pudo leer imagen izquierda:
{left_path}")

    if imgR is None:
        raise FileNotFoundError(f"No se pudo leer imagen derecha:
{right_path}")

```

```

    if imgL.shape != imgR.shape:
        raise RuntimeError(
            f"Las imágenes izquierda y derecha no tienen el mismo
tamaño: "
            f"{left_path} {imgL.shape} vs {right_path}
{imgR.shape}"
        )

    start_time = time.perf_counter()

    disp_fixed = matcher.compute(imgL, imgR)

    end_time = time.perf_counter()

    disparity_time = end_time - start_time

    # OpenCV devuelve la disparidad en formato fijo escalado por
    16.
    # Esta conversión queda FUERA del tiempo medido, igual que en
    PSMNet
    # quedan fuera pequeñas operaciones posteriores a model(imgL,
    imgR).
    pred_disp = disp_fixed.astype(np.float32) / 16.0

    return pred_disp, disparity_time

# =====
# 6. MÉTRICAS ANÁLOGAS A EVALUACIÓN PSMNet
# =====

def compute_basic_metrics(pred_disp_np, gt_disp_np, maxdisp=192):
    """
    Calcula:
        - D1-all
        - EPE
        - AbsRel
        - delta1

    Misma filosofía que en la evaluación PSMNet:

```

```

    valid = (gt > 0) & (gt < maxdisp)

No se filtra por predicción válida.
Si StereoSGBM produce disparidad negativa o cero, se penaliza.
"""

pred_disp = torch.from_numpy(pred_disp_np).float()
gt_disp = torch.from_numpy(gt_disp_np).float()

if pred_disp.shape != gt_disp.shape:
    raise RuntimeError(
        f"Predicción y GT tienen formas diferentes: "
        f"pred={tuple(pred_disp.shape)}
gt={tuple(gt_disp.shape)}"
    )

valid = (gt_disp > 0) & (gt_disp < maxdisp)

valid_pixels = int(valid.sum().item())

if valid_pixels == 0:
    return {
        "dl_all": np.nan,
        "epe": np.nan,
        "abs_rel": np.nan,
        "delta1": np.nan,
        "valid_pixels": 0,
    }

pred_valid = pred_disp[valid]
gt_valid = gt_disp[valid]

abs_err = torch.abs(pred_valid - gt_valid)

if abs_err.numel() == 0 or not torch.isfinite(abs_err).all():
    return {
        "dl_all": np.nan,
        "epe": np.nan,
        "abs_rel": np.nan,
        "delta1": np.nan,
    }

```

```

        "valid_pixels": valid_pixels,
    }

    epe = abs_err.mean().item()

    d1_all = (
        ((abs_err > 3.0) & (abs_err > 0.05 * gt_valid))
        .float()
        .mean()
        .item()
    )

    abs_rel = (
        abs_err / torch.clamp(gt_valid, min=1e-6)
    ).mean().item()

    pred_safe = torch.clamp(pred_valid, min=1e-6)
    gt_safe = torch.clamp(gt_valid, min=1e-6)

    ratio = torch.maximum(pred_safe / gt_safe, gt_safe / pred_safe)
    delta1 = (ratio < 1.25).float().mean().item()

    return {
        "d1_all": d1_all,
        "epe": epe,
        "abs_rel": abs_rel,
        "delta1": delta1,
        "valid_pixels": valid_pixels,
    }

# =====
# 7. GUARDADO CSV
# =====

def safe_float(x):
    if x is None:
        return "nan"

    if isinstance(x, (float, np.floating)):

```

```

        if not np.isfinite(x):
            return "nan"
        return f"{float(x):.6f}"

    return x

def save_metrics_per_image(rows, out_path):
    with open(out_path, "w", newline="") as f:
        writer = csv.writer(f)

        writer.writerow([
            "sample",
            "dl_all",
            "epe",
            "abs_rel",
            "delta1",
            "valid_pixels",
        ])

    for r in rows:
        writer.writerow([
            r["sample"],
            safe_float(r["dl_all"]),
            safe_float(r["epe"]),
            safe_float(r["abs_rel"]),
            safe_float(r["delta1"]),
            r["valid_pixels"],
        ])

def save_metrics_summary(rows, out_path):
    metrics = ["dl_all", "epe", "abs_rel", "delta1"]

    with open(out_path, "w", newline="") as f:
        writer = csv.writer(f)

        writer.writerow(["metric", "value"])
        writer.writerow(["num_samples", len(rows)])

```



```

        writer.writerow(["total_valid_pixels",
sum(r["valid_pixels"] for r in rows)])

    for m in metrics:
        vals = [r[m] for r in rows if np.isfinite(r[m])]
        mean_val = float(np.mean(vals)) if vals else np.nan
        writer.writerow([m, safe_float(mean_val)])

def save_time_metrics(time_info, out_path):
    with open(out_path, "w", newline="") as f:
        writer = csv.writer(f)

        writer.writerow(["metric", "value"])
        writer.writerow(["num_images", time_info["num_images"]])
        writer.writerow(["total_disparity_time_sec",
safe_float(time_info["total_disparity_time_sec"])]])
        writer.writerow(["mean_disparity_time_per_image_sec",
safe_float(time_info["mean_disparity_time_per_image_sec"])]])
        writer.writerow(["fps", safe_float(time_info["fps"])]])
        writer.writerow(["timing_definition",
time_info["timing_definition"]])
        writer.writerow(["timing_included",
time_info["timing_included"]])
        writer.writerow(["timing_excluded",
time_info["timing_excluded"]])

# =====
# 8. GUARDADO DE MAPAS DE DISPARIDAD
# =====

def imwrite_checked(path, image):
    ok = cv2.imwrite(str(path), image)

    if not ok:
        raise RuntimeError(f"No se pudo guardar imagen: {path}")

def prepare_disparity_map_dirs(output_dir):

```

```

base_dir = Path(output_dir) / "MapasDisparidad"

dirs = {
    "base": base_dir,
    "pred_npy": base_dir / "PredNpy",
    "pred_kitti_png": base_dir / "PredKittiPng_x256",
    "pred_visual_png": base_dir / "PredVisualPng",
    "gt_visual_png": base_dir / "GtVisualPng",
    "error_visual_png": base_dir / "ErrorVisualPng",
}

for d in dirs.values():
    ensure_dir(d)

return dirs


def should_save_visual_for_index(idx, config):
    max_visual = config.get("max_visual_maps_to_save", None)

    if max_visual is None:
        return True

    return idx < int(max_visual)


def make_disparity_visual(disparity, maxdisp=192, valid_mask=None):
    """
    Convierte un mapa de disparidad a imagen visual uint8 con mapa
    de color.

    Esta imagen es SOLO para visualizar.
    No debe usarse para calcular métricas.
    """

    if valid_mask is None:
        valid_mask = np.isfinite(disparity) & (disparity > 0.0)

    disparity_clean = np.nan_to_num(disparity, nan=0.0, posinf=0.0,
                                     neginf=0.0)

```

```

disp_clip = np.clip(disp_clean, 0.0, float(maxdisp))

visual_gray = ((disp_clip / float(maxdisp)) *
255.0).astype(np.uint8)
visual_gray[~valid_mask] = 0

if hasattr(cv2, "COLORMAP_TURBO"):
    colormap = cv2.COLORMAP_TURBO
else:
    colormap = cv2.COLORMAP_JET

visual_color = cv2.applyColorMap(visual_gray, colormap)

return visual_color

def make_error_visual(pred_disp, gt_disp, valid_mask,
error_visual_max_px=20.0):
    """
    Crea una visualización del error absoluto |pred - GT|.

    Esta imagen es SOLO para visualizar.
    No debe usarse para calcular métricas.
    """

    abs_err = np.abs(pred_disp - gt_disp)
    abs_err = np.nan_to_num(abs_err, nan=0.0, posinf=0.0,
neginf=0.0)
    abs_err_clip = np.clip(abs_err, 0.0,
float(error_visual_max_px))

    visual_gray = ((abs_err_clip / float(error_visual_max_px)) *
255.0).astype(np.uint8)
    visual_gray[~valid_mask] = 0

    if hasattr(cv2, "COLORMAP_TURBO"):
        colormap = cv2.COLORMAP_TURBO
    else:
        colormap = cv2.COLORMAP_JET

```

```

visual_color = cv2.applyColorMap(visual_gray, colormap)

return visual_color

def save_pred_as_kitti_png(pred_disp, out_path, maxdisp=192):
    """
    Guarda la predicción como PNG uint16 estilo KITTI:
        valor_guardado = disparidad * 256

    Nota:
        - Las disparidades negativas o no finitas se guardan como 0.
        - Las disparidades mayores que maxdisp se saturan a maxdisp.
        - Para conservar exactamente la salida original, se guarda
        también .npy.
    """

    pred_clean = np.nan_to_num(pred_disp, nan=0.0, posinf=0.0,
                                neginf=0.0)
    pred_clean = np.clip(pred_clean, 0.0, float(maxdisp))

    pred_uint16 = np.round(pred_clean * 256.0).astype(np.uint16)

    imwrite_checked(out_path, pred_uint16)

def save_disparity_maps_for_sample(
    pred_disp,
    gt_disp,
    sample_name,
    idx,
    map_dirs,
    config,
):
    if not config.get("save_disparity_maps", True):
        return

    sample_stem = Path(sample_name).stem

    valid_gt_mask = (

```

```

        np.isfinite(gt_disp)
        & (gt_disp > 0.0)
        & (gt_disp < float(config["maxdisp"]))
    )

    # 1. Guardado de la predicción en píxeles reales
    if config.get("save_pred_npy", True):
        npy_path = map_dirs["pred_npy"] /
f"{sample_stem}_pred_disp_px.npy"
        np.save(npy_path, pred_disp.astype(np.float32))

    # 2. Guardado como PNG uint16 estilo KITTI, escalado x256
    if config.get("save_pred_kitti_png", True):
        kitti_png_path = map_dirs["pred_kitti_png"] /
f"{sample_stem}_pred_disp_x256.png"
        save_pred_as_kitti_png(
            pred_disp=pred_disp,
            out_path=kitti_png_path,
            maxdisp=config["maxdisp"],
        )

    # 3. Guardado visual
    if config.get("save_visual_maps", True) and
should_save_visual_for_index(idx, config):
        pred_valid_mask = np.isfinite(pred_disp) & (pred_disp >
0.0)

        pred_visual = make_disparity_visual(
            disp=pred_disp,
            maxdisp=config["maxdisp"],
            valid_mask=pred_valid_mask,
        )

        pred_visual_path = map_dirs["pred_visual_png"] /
f"{sample_stem}_pred_visual.png"
        imwrite_checked(pred_visual_path, pred_visual)

    if config.get("save_gt_visual_maps", True):
        gt_visual = make_disparity_visual(
            disp=gt_disp,

```

```

        maxdisp=config["maxdisp"],
        valid_mask=valid_gt_mask,
    )

    gt_visual_path = map_dirs["gt_visual_png"] /
f"{sample_stem}_gt_visual.png"
    imwrite_checked(gt_visual_path, gt_visual)

    if config.get("save_error_visual_maps", True):
        error_visual = make_error_visual(
            pred_disp=pred_disp,
            gt_disp=gt_disp,
            valid_mask=valid_gt_mask,
            error_visual_max_px=config["error_visual_max_px"],
        )

        error_visual_path = map_dirs["error_visual_png"] /
f"{sample_stem}_error_visual.png"
        imwrite_checked(error_visual_path, error_visual)

# =====
# 9. EVALUACIÓN MÉTODO CLÁSICO
# =====

def evaluate_classic_method(config):
    ensure_dir(config["output_dir"])

    map_dirs = prepare_disparity_map_dirs(config["output_dir"])

    print("=====
    ==")
    print("EVALUACIÓN MÉTODO CLÁSICO - StereoSGBM")

    print("=====
    ==")
    print("Versión OpenCV:", cv2.__version__)
    print("Nota: aunque Colab tenga GPU activa,
    cv2.StereoSGBM_create normalmente se ejecuta en CPU.")

```

```

print("=====
====")

kitti2015_root = prepare_kitti2015(config)

left_paths, right_paths, gt_paths, sample_names =
build_kitti2015_final_eval_lists_second_half(
    kitti2015_root
)

matcher = create_sgbm_matcher(config)

per_image_rows = []

total_disparity_time = 0.0
measured_images = 0

skipped_nonfinite = 0
skipped_invalid = 0

for idx, (left_path, right_path, gt_path, sample_name) in
enumerate(
    zip(left_paths, right_paths, gt_paths, sample_names)
):
    pred_disp, disparity_time = compute_classic_disparity_sgbm(
        left_path=left_path,
        right_path=right_path,
        matcher=matcher,
    )

    # El tiempo se suma si matcher.compute() se ha ejecutado
correctamente,
    # aunque luego alguna métrica pudiera no ser válida.
    total_disparity_time += disparity_time
    measured_images += 1

gt_disp = read_kitti_disparity_png(gt_path)

# Guardado de mapas fuera del tiempo medido

```

```

save_disparity_maps_for_sample(
    pred_disp=pred_disp,
    gt_disp=gt_disp,
    sample_name=sample_name,
    idx=idx,
    map_dirs=map_dirs,
    config=config,
)

if not np.isfinite(pred_disp).all() or not
np.isfinite(gt_disp).all():
    skipped_nonfinite += 1
    print(f"[WARN] {sample_name}: predicción o GT no
finita. Se omite en métricas.")
    continue

basic = compute_basic_metrics(
    pred_disp_np=pred_disp,
    gt_disp_np=gt_disp,
    maxdisp=config["maxdisp"],
)

if not (
    np.isfinite(basic["d1_all"])
    and np.isfinite(basic["epe"])
    and np.isfinite(basic["abs_rel"])
    and np.isfinite(basic["delta1"])
):
    skipped_invalid += 1
    print(f"[WARN] {sample_name}: métricas inválidas. Se
omite en métricas.")
    continue

per_image_rows.append({
    "sample": sample_name,
    "d1_all": basic["d1_all"],
    "epe": basic["epe"],
    "abs_rel": basic["abs_rel"],
    "delta1": basic["delta1"],
    "valid_pixels": basic["valid_pixels"],

```



```

    })

    print(
        f"{idx + 1:03d}/{len(sample_names)} | {sample_name} | "
        f"D1-all={basic['d1_all']:.6f} | "
        f"EPE={basic['epe']:.6f} | "
        f"AbsRel={basic['abs_rel']:.6f} | "
        f"delta1={basic['delta1']:.6f} | "
        f"t_disp={disparity_time:.4f}s"
    )

    if len(per_image_rows) == 0:
        raise RuntimeError("La evaluación no produjo ninguna
métrica válida.")

    mean_time = (
        total_disparity_time / measured_images
        if measured_images > 0
        else np.nan
    )

    fps = (
        measured_images / total_disparity_time
        if total_disparity_time > 0
        else np.nan
    )

    time_info = {
        "num_images": measured_images,
        "total_disparity_time_sec": total_disparity_time,
        "mean_disparity_time_per_image_sec": mean_time,
        "fps": fps,
        "timing_definition": "Tiempo de cálculo principal de
disparidad.",
        "timing_included": "Solo matcher.compute(imgL, imgR).",
        "timing_excluded": (
            "Lectura de imágenes, conversión fixed-point a píxeles,
lectura de GT, "
            "cálculo de métricas, guardado de mapas, guardado de
CSV y creación del matcher."
        )
    }

```

```

    ),
}

metrics_per_image_path = os.path.join(config["output_dir"],
"metrics_per_image.csv")
metrics_summary_path = os.path.join(config["output_dir"],
"metrics_summary.csv")
time_metrics_path = os.path.join(config["output_dir"],
"metricas_tiempo.csv")
evaluation_config_path = os.path.join(config["output_dir"],
"evaluation_config.json")

save_metrics_per_image(per_image_rows, metrics_per_image_path)
save_metrics_summary(per_image_rows, metrics_summary_path)
save_time_metrics(time_info, time_metrics_path)

evaluation_config = {
    "metodo": "StereoSGBM OpenCV",
    "descripcion": "Método clásico de correspondencia estéreo
densa sin CNN.",
    "dataset": "KITTI2015 training segunda mitad",
    "kitti2015_split": "índices 100-199, imágenes _10 y _11",
    "rectificacion": "No se realiza rectificación epipolar
adicional porque KITTI2015 ya proporciona pares estéreo
rectificados.",
    "metricas": {
        "dl_all": "Porcentaje de píxeles válidos con error
absoluto > 3 px y error relativo > 5%.",
        "epe": "Error absoluto medio de disparidad en
píxeles.",
        "abs_rel": "Media de |pred-GT| / GT sobre píxeles
válidos.",
        "delta1": "Porcentaje de píxeles con max(pred/GT,
GT/pred) < 1.25.",
    },
    "valid_mask": "gt_disp > 0 y gt_disp < maxdisp",
    "maxdisp": config["maxdisp"],
    "output_dir": config["output_dir"],
    "opencv_version": cv2.__version__,
    "hardware_note": (

```

"Aunque Colab tenga GPU activa, cv2.StereoSGBM_create normalmente se ejecuta en CPU "

"salvo que OpenCV esté compilado con soporte CUDA y se use explícitamente una API CUDA. "

"La comparación temporal iguala el criterio de inicio y fin del conteo, pero no necesariamente el hardware."

```

    ),
    "sgbm_parameters": {
        "min_disparity": config["sgbm_min_disparity"],
        "num_disparities": config["sgbm_num_disparities"],
        "block_size": config["sgbm_block_size"],
        "uniqueness_ratio": config["sgbm_uniqueness_ratio"],
        "speckle_window_size":
config["sgbm_speckle_window_size"],
        "speckle_range": config["sgbm_speckle_range"],
        "disp12_max_diff": config["sgbm_disp12_max_diff"],
        "pre_filter_cap": config["sgbm_pre_filter_cap"],
        "mode": "STEREO_SGBM_MODE_SGBM_3WAY",
    },
    "time_metrics": {
        "name": "tiempo de cálculo principal de disparidad",
        "psmnet_reference": "En PSMNet se mide solo model(imgL,
imgR).",
        "classic_method": "En StereoSGBM se mide solo
matcher.compute(imgL, imgR).",
        "included": [
            "matcher.compute(imgL, imgR)"
        ],
        "excluded": [
            "lectura de imágenes",
            "conversión de fixed-point OpenCV a píxeles
reales",
            "lectura de GT",
            "cálculo de métricas",
            "guardado de CSV",
            "guardado de mapas de disparidad",
            "creación del matcher SGBM",
        ],
    },
    "saved_disparity_maps": {

```

```

        "base_dir": str(map_dirs["base"]),
        "pred_npy": str(map_dirs["pred_npy"]),
        "pred_kitti_png_x256": str(map_dirs["pred_kitti_png"]),
        "pred_visual_png": str(map_dirs["pred_visual_png"]),
        "gt_visual_png": str(map_dirs["gt_visual_png"]),
        "error_visual_png": str(map_dirs["error_visual_png"]),
        "important_note": (
            "Los PNG visuales están normalizados solo para
inspección visual. "
            "Las métricas se calculan siempre con pred_disp en
píxeles reales, no con las visualizaciones."
        ),
    },
}

with open(evaluation_config_path, "w") as f:
    json.dump(evaluation_config, f, indent=2)

print("\n=====
=====")
print("EVALUACIÓN MÉTODO CLÁSICO FINALIZADA")

print("=====
=====")
print(f"CSV por imagen guardado en: {metrics_per_image_path}")
print(f"CSV resumen guardado en: {metrics_summary_path}")
print(f"CSV métricas tiempo guardado en: {time_metrics_path}")
print(f"Config guardada en: {evaluation_config_path}")
print(f"Mapas de disparidad guardados en: {map_dirs['base']}")
print(f"Muestras omitidas por NaN/Inf en métricas:
{skipped_nonfinite}")
print(f"Muestras omitidas por métricas inválidas:
{skipped_invalid}")
print(f"Tiempo total cálculo principal de disparidad:
{total_disparity_time:.4f} s")
print(f"Tiempo medio por imagen: {mean_time:.4f} s")
print(f"FPS aproximados: {fps:.4f}")

```

```

print("=====
===")

return {
    "metrics_per_image": metrics_per_image_path,
    "metrics_summary": metrics_summary_path,
    "metricas_tiempo": time_metrics_path,
    "evaluation_config": evaluation_config_path,
    "disparity_maps_dir": str(map_dirs["base"]),
    "time_info": time_info,
}

# =====
# 10. EJECUCIÓN
# =====

try:
    resultados_metodo_clasico = evaluate_classic_method(CONFIG)
    print(resultados_metodo_clasico)

finally:
    if CONFIG.get("auto_disconnect", False):
        from google.colab import runtime
        runtime.unassign()

```

7.2. ANEXO 2: FINE-TUNING DE TODOS LOS EXPERIMENTOS

```
# PSMNet fine-tuning en Colab para TFG

# En este script se realizó el fine-tuning del modelo PSMNet
# utilizando el dataset KITTI 2012 completo para entrenamiento y
# KITTI 2015 primera mitad para seleccionar la mejor época.
# Guarda las métricas de todas las épocas, los pesos solo cuando
# se alcanza un nuevo mínimo de D1-all, y además guarda siempre
# los pesos de la última época ejecutada.

# ENTORNO

import os
import sys
import json
import time
import random
import zipfile
import shutil
import subprocess
import csv
from pathlib import Path

import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from torch.utils.data import DataLoader
from google.colab import runtime

print("Comprobando entorno GPU...")
nvidia_smi_path = shutil.which("nvidia-smi")
if nvidia_smi_path is not None:
    try:
        subprocess.run([nvidia_smi_path], check=False)
```

```

except Exception as e:
    print(f"[WARN] No se pudo ejecutar nvidia-smi: {e}")
else:
    print("[WARN] nvidia-smi no está disponible en este entorno.")

from google.colab import drive
drive.mount('/content/drive', force_remount=True)

!pip -q install gdown

# CONFIGURACIÓN

CONFIG = {
    "kitti2012_zip_drive":
    "/content/drive/MyDrive/TFG/Datasets/KITTI2012/data_stereo_flow.
zip",
    "kitti2015_zip_drive":
    "/content/drive/MyDrive/TFG/Datasets/KITTI2015/data_scene_flow.z
ip",

    "pretrained_drive_path":
    "/content/drive/MyDrive/TFG/Entrenamiento/PesosPreentrenamiento/pretra
ined_sceneflow_new.tar",
    "pretrained_url":
    "https://drive.google.com/uc?id=1NDKrWHkgwMKtDwynXVU12emK3G5d5kk
p",

    "seed": 42,
    "maxdisp": 192,
    "batch_size_train": 4,
    "batch_size_eval": 1,
    "lr": 1e-3,
    "min_delta": 1e-4,
    "max_epochs": 300,
    "num_workers": 2,
}

```

```
# UTILIDADES GENERALES
```

```
def set_all_seeds(seed: int):  
    random.seed(seed)  
    np.random.seed(seed)  
    torch.manual_seed(seed)  
    torch.cuda.manual_seed_all(seed)  
    os.environ["PYTHONHASHSEED"] = str(seed)  
    torch.backends.cudnn.deterministic = True  
    torch.backends.cudnn.benchmark = False
```

```
def ensure_dir(path):  
    Path(path).mkdir(parents=True, exist_ok=True)
```

```
def remove_dir_if_exists(path: str):  
    p = Path(path)  
    if p.exists():  
        shutil.rmtree(p)
```

```
def assert_gpu_available():  
    if not torch.cuda.is_available():  
        raise RuntimeError(  
            "No hay GPU activa en Colab. Activa la GPU antes de  
            ejecutar este entrenamiento."  
        )
```

```
def copy_zip_from_drive_to_content(src_drive_path: str, dst_dir:  
    str = "/content/datasets_zip"):  
    ensure_dir(dst_dir)  
    src = Path(src_drive_path)  
  
    if not src.exists():  
        raise FileNotFoundError(f"No existe el ZIP en Drive:  
{src_drive_path}")  
  
    dst = Path(dst_dir) / src.name
```



```

print(f"Copiando {src} -> {dst}")
shutil.copy2(src, dst)

if not dst.exists() or dst.stat().st_size == 0:
    raise RuntimeError(f"Error al copiar el ZIP: {dst}")

return str(dst)

def unzip_file(zip_path: str, extract_to: str,
               clean_before_extract: bool = True):
    if clean_before_extract:
        remove_dir_if_exists(extract_to)

    ensure_dir(extract_to)

    print(f"Descomprimiendo {zip_path} en {extract_to} ...")

    try:
        with zipfile.ZipFile(zip_path, 'r') as zf:
            bad_member = zf.testzip()
            if bad_member is not None:
                raise RuntimeError(f"ZIP corrupto o incompleto.
Primer archivo problemático: {bad_member}")
            zf.extractall(extract_to)
    except zipfile.BadZipFile:
        raise RuntimeError(f"El archivo no es un ZIP válido:
{zip_path}")

    if not Path(extract_to).exists():
        raise RuntimeError(f"No se creó la carpeta de extracción:
{extract_to}")

    return extract_to

def find_kitti_training_root(search_root: str, year: int):
    search_root = Path(search_root)

    if year == 2012:

```

```

        required = {"colored_0", "colored_1", "disp_occ"}
    elif year == 2015:
        required = {"image_2", "image_3", "disp_occ_0"}
    else:
        raise ValueError("year debe ser 2012 o 2015")

    for root, dirs, files in os.walk(search_root):
        if required.issubset(set(dirs)):
            return root + "/"

    raise FileNotFoundError(f"No se encontró la raíz training de
KITTI {year} dentro de {search_root}")

def get_pretrained_checkpoint(config,
local_fallback_path="/content/pretrained_sceneflow_new.tar"):
    drive_path = config.get("pretrained_drive_path", None)

    if drive_path is not None:
        drive_path = Path(drive_path)
        if drive_path.exists() and drive_path.stat().st_size > 0:
            print(f"Usando checkpoint preentrenado desde Drive:
{drive_path}")
            return str(drive_path)

    print("No se encontró checkpoint válido en Drive. Se descargará
con gdown...")

    if os.path.exists(local_fallback_path):
        os.remove(local_fallback_path)

    !gdown "{config['pretrained_url']}" -O "{local_fallback_path}"

    if not os.path.exists(local_fallback_path) or
os.path.getsize(local_fallback_path) == 0:
        raise FileNotFoundError("No se pudo obtener correctamente
el checkpoint preentrenado.")

    return local_fallback_path

```

```

def prepare_all_data(config):
    local_zip_dir = "/content/datasets_zip"
    local_extract_dir = "/content/datasets"

    k2012_zip =
copy_zip_from_drive_to_content(config["kitti2012_zip_drive"],
local_zip_dir)
    k2015_zip =
copy_zip_from_drive_to_content(config["kitti2015_zip_drive"],
local_zip_dir)

    k2012_extract = unzip_file(k2012_zip,
os.path.join(local_extract_dir, "kitti2012"),
clean_before_extract=True)
    k2015_extract = unzip_file(k2015_zip,
os.path.join(local_extract_dir, "kitti2015"),
clean_before_extract=True)

    kitti2012_root = find_kitti_training_root(k2012_extract, 2012)
    kitti2015_root = find_kitti_training_root(k2015_extract, 2015)

    pretrained_path = get_pretrained_checkpoint(config)

    return {
        "kitti2012_root": kitti2012_root,
        "kitti2015_root": kitti2015_root,
        "pretrained_path": pretrained_path,
    }

# CLONACIÓN DE REPOSITORIO OFICIAL

def clone_and_import_psmnet(repo_dir="/content/PSMNet"):
    print("Preparando repositorio oficial de PSMNet...")

    os.chdir("/content")
    !rm -rf /content/PSMNet
    !git clone https://github.com/JiaRenChang/PSMNet.git

```

```

if not Path(repo_dir).exists():
    raise RuntimeError("No se pudo clonar el repositorio
PSMNet.")

os.chdir("/content/PSMNet")

repo_dir_str = str(Path(repo_dir).resolve())
dataloader_dir_str = str((Path(repo_dir) /
"dataloader").resolve())
models_dir_str = str((Path(repo_dir) / "models").resolve())

for p in [repo_dir_str, dataloader_dir_str, models_dir_str]:
    if p not in sys.path:
        sys.path.insert(0, p)

expected_files = [
    Path(repo_dir) / "models" / "__init__.py",
    Path(repo_dir) / "dataloader" / "KITTIILoader.py",
    Path(repo_dir) / "dataloader" / "preprocess.py",
]

for f in expected_files:
    if not f.exists():
        raise FileNotFoundError(f"Falta archivo esperado del
repo oficial: {f}")

try:
    from models import stackhourglass
except Exception as e:
    raise RuntimeError(f"Error al importar stackhourglass desde
models: {e}")

try:
    from dataloader.KITTIILoader import myImageFloder
except Exception as e:
    raise RuntimeError(
        "Error al importar myImageFloder desde
dataloader.KITTIILoader. "
        f"Error original: {e}"
    )

```

```

    print("Repositorio PSMNet listo e imports verificados.")
    return stackhourglass, myImageFloder

stackhourglass, myImageFloder = clone_and_import_psmnet()

# CAPAS / MODOS

def set_requires_grad(module, flag: bool):
    for p in module.parameters():
        p.requires_grad = flag

def configure_trainable_layers(model: nn.Module, mode: str):
    """
    Modos preparados:
    - todas_capas
    - intermedias_finales
    - solo_finales
    """
    if mode not in {"todas_capas", "intermedias_finales",
"solo_finales"}:
        raise ValueError(f"Modo no soportado: {mode}")

    for p in model.parameters():
        p.requires_grad = False

    if mode == "todas_capas":
        for p in model.parameters():
            p.requires_grad = True

    elif mode == "intermedias_finales":
        set_requires_grad(model.feature_extraction.layer3, True)
        set_requires_grad(model.feature_extraction.layer4, True)

        set_requires_grad(model.feature_extraction.branch1, True)
        set_requires_grad(model.feature_extraction.branch2, True)

```

```

set_requires_grad(model.feature_extraction.branch3, True)
set_requires_grad(model.feature_extraction.branch4, True)
set_requires_grad(model.feature_extraction.lastconv, True)

set_requires_grad(model.dres0, True)
set_requires_grad(model.dres1, True)
set_requires_grad(model.dres2, True)
set_requires_grad(model.dres3, True)
set_requires_grad(model.dres4, True)
set_requires_grad(model.classif1, True)
set_requires_grad(model.classif2, True)
set_requires_grad(model.classif3, True)

elif mode == "solo_finales":
    set_requires_grad(model.feature_extraction.branch1, True)
    set_requires_grad(model.feature_extraction.branch2, True)
    set_requires_grad(model.feature_extraction.branch3, True)
    set_requires_grad(model.feature_extraction.branch4, True)
    set_requires_grad(model.feature_extraction.lastconv, True)

    set_requires_grad(model.dres0, True)
    set_requires_grad(model.dres1, True)
    set_requires_grad(model.dres2, True)
    set_requires_grad(model.dres3, True)
    set_requires_grad(model.dres4, True)
    set_requires_grad(model.classif1, True)
    set_requires_grad(model.classif2, True)
    set_requires_grad(model.classif3, True)

trainable, frozen = [], []

for name, p in model.named_parameters():
    if p.requires_grad:
        trainable.append(name)
    else:
        frozen.append(name)

return {
    "trainable": trainable,
    "frozen": frozen,

```

```

    }

# LISTAS DE DATOS

def build_kitti2012_train_lists_all_available(kitti2012_root: str):
    left_dir = Path(kitti2012_root) / "colored_0"
    right_dir = Path(kitti2012_root) / "colored_1"
    disp_dir = Path(kitti2012_root) / "disp_occ"

    left_names = {p.name for p in left_dir.glob("*.png")}
    right_names = {p.name for p in right_dir.glob("*.png")}
    disp_names = {p.name for p in disp_dir.glob("*.png")}

    common = sorted(left_names & right_names & disp_names)

    if len(common) == 0:
        raise RuntimeError("No se encontraron muestras válidas en KITTI 2012.")

    left = [str(left_dir / n) for n in common]
    right = [str(right_dir / n) for n in common]
    disp = [str(disp_dir / n) for n in common]

    print(f"KITTI 2012 entrenamiento: {len(common)} muestras con GT.")
    return left, right, disp, common

def
build_kitti2015_selection_lists_fixed_first_half(kitti2015_root:
str):
    left_dir = Path(kitti2015_root) / "image_2"
    right_dir = Path(kitti2015_root) / "image_3"
    disp_dir = Path(kitti2015_root) / "disp_occ_0"

    requested = []

    for i in range(100):
        requested.append(f"{i:06d}_10.png")

```

```

        requested.append(f"{i:06d}_11.png")

    left_names = {p.name for p in left_dir.glob("*.png")}
    right_names = {p.name for p in right_dir.glob("*.png")}
    disp_names = {p.name for p in disp_dir.glob("*.png")}

    common = []

    for n in requested:
        if n in left_names and n in right_names and n in
disp_names:
            common.append(n)

    if len(common) == 0:
        raise RuntimeError("No se encontraron muestras válidas para
selección de época en KITTI 2015.")

    left = [str(left_dir / n) for n in common]
    right = [str(right_dir / n) for n in common]
    disp = [str(disp_dir / n) for n in common]

    print(f"KITTI 2015 model selection: {len(common)} muestras.")
    return left, right, disp, common

def save_list_txt(path, names):
    with open(path, "w") as f:
        for n in names:
            f.write(f"{n}\n")

# LOADERS

def make_loaders(kitti2012_root, kitti2015_root,
    batch_size_train=2, batch_size_eval=1, num_workers=2):
    train_left, train_right, train_disp, train_names =
build_kitti2012_train_lists_all_available(kitti2012_root)
    sel_left, sel_right, sel_disp, sel_names =
build_kitti2015_selection_lists_fixed_first_half(kitti2015_root)

```



```

train_dataset = myImageFloder(train_left, train_right,
train_disp, training=True)
selection_dataset = myImageFloder(sel_left, sel_right,
sel_disp, training=False)

train_loader = DataLoader(
    train_dataset,
    batch_size=batch_size_train,
    shuffle=True,
    num_workers=num_workers,
    drop_last=False,
    pin_memory=True,
)

selection_loader = DataLoader(
    selection_dataset,
    batch_size=batch_size_eval,
    shuffle=False,
    num_workers=num_workers,
    drop_last=False,
    pin_memory=True,
)

return train_loader, selection_loader, train_names, sel_names

```

MÉTRICAS

```

def compute_metrics_from_prediction(pred_disp, gt_disp,
maxdisp=192):
    valid = (gt_disp > 0) & (gt_disp < maxdisp)

    if valid.sum().item() == 0:
        return {
            "dl_all": np.nan,
            "epe": np.nan,
            "abs_rel": np.nan,
            "delta1": np.nan,
        }

```

```

pred_valid = pred_disp[valid]
gt_valid = gt_disp[valid]

abs_err = torch.abs(pred_valid - gt_valid)

if abs_err.numel() == 0:
    return {
        "dl_all": np.nan,
        "epe": np.nan,
        "abs_rel": np.nan,
        "delta1": np.nan,
    }

if not torch.isfinite(abs_err).all():
    return {
        "dl_all": np.nan,
        "epe": np.nan,
        "abs_rel": np.nan,
        "delta1": np.nan,
    }

epe = abs_err.mean().item()

dl = (
    ((abs_err > 3.0) & (abs_err > 0.05 * gt_valid))
    .float()
    .mean()
    .item()
)

abs_rel = (abs_err / gt_valid).mean().item()

pred_safe = torch.clamp(pred_valid, min=1e-6)
gt_safe = torch.clamp(gt_valid, min=1e-6)

ratio = torch.maximum(pred_safe / gt_safe, gt_safe / pred_safe)
delta1 = (ratio < 1.25).float().mean().item()

if (
    not np.isfinite(epe)

```

```

        or not np.isfinite(dl)
        or not np.isfinite(abs_rel)
        or not np.isfinite(delta1)
    ):
        return {
            "dl_all": np.nan,
            "epe": np.nan,
            "abs_rel": np.nan,
            "delta1": np.nan,
        }

    return {
        "dl_all": dl,
        "epe": epe,
        "abs_rel": abs_rel,
        "delta1": delta1,
    }

def save_history_as_csv(history_rows, out_path):
    with open(out_path, "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["epoch", "train_loss", "dl_all", "epe",
            "abs_rel", "delta1"])

        for row in history_rows:
            writer.writerow([
                row["epoch"],
                "nan" if not np.isfinite(row["train_loss"]) else
f"{row['train_loss']:.6f}",
                "nan" if not np.isfinite(row["dl_all"]) else
f"{row['dl_all']:.6f}",
                "nan" if not np.isfinite(row["epe"]) else
f"{row['epe']:.6f}",
                "nan" if not np.isfinite(row["abs_rel"]) else
f"{row['abs_rel']:.6f}",
                "nan" if not np.isfinite(row["delta1"]) else
f"{row['delta1']:.6f}",
            ])

```

```
# CHECKPOINT PREENTRENADO
```

```
def load_official_checkpoint(model: nn.Module, ckpt_path: str,
                             device: torch.device):
    if not os.path.exists(ckpt_path) or os.path.getsize(ckpt_path)
    == 0:
        raise FileNotFoundError(f"Checkpoint no encontrado o vacío:
        {ckpt_path}")

    ckpt = torch.load(ckpt_path, map_location=device)
    state = ckpt["state_dict"] if isinstance(ckpt, dict) and
    "state_dict" in ckpt else ckpt

    if not isinstance(state, dict):
        raise RuntimeError("El checkpoint no contiene un state_dict
        válido.")

    clean_state = {}

    for k, v in state.items():
        clean_key = k.replace("module.", "")
        clean_state[clean_key] = v

    model_state = model.state_dict()

    missing_keys = sorted(set(model_state.keys()) -
    set(clean_state.keys()))
    unexpected_keys = sorted(set(clean_state.keys()) -
    set(model_state.keys()))

    shape_mismatch = []

    for k in set(model_state.keys()) & set(clean_state.keys()):
        if model_state[k].shape != clean_state[k].shape:
            shape_mismatch.append((k, tuple(clean_state[k].shape),
            tuple(model_state[k].shape)))

    if missing_keys:
        msg = "\n".join(missing_keys[:20])
```

```

        raise RuntimeError(f"Faltan claves del checkpoint al cargar
el modelo. Primeras:\n{msg}")

    if unexpected_keys:
        msg = "\n".join(unexpected_keys[:20])
        raise RuntimeError(f"Hay claves inesperadas en el
checkpoint. Primeras:\n{msg}")

    if shape_mismatch:
        preview = "\n".join([f"{k}: ckpt{a} vs model{b}" for k, a,
b in shape_mismatch[:20]])
        raise RuntimeError(f"Hay tensores con forma incompatible en
el checkpoint. Primeros:\n{preview}")

    model.load_state_dict(clean_state, strict=True)

    print("Checkpoint cargado correctamente y verificado de forma
estricta.")
    return model

def count_trainable_params(model):
    return sum(p.numel() for p in model.parameters() if
p.requires_grad)

# TRAIN / EVAL

def train_one_epoch(model, loader, optimizer, device, maxdisp=192):
    model.train()

    losses = []
    skipped_batches_invalid_gt = 0
    skipped_batches_nonfinite = 0

    for batch_idx, (imgL, imgR, dispL) in enumerate(loader):
        imgL = imgL.float().to(device, non_blocking=True)
        imgR = imgR.float().to(device, non_blocking=True)
        dispL = dispL.float().to(device, non_blocking=True)

```

```

        if not torch.isfinite(imgL).all() or not
torch.isfinite(imgR).all() or not torch.isfinite(displ).all():
            skipped_batches_nonfinite += 1
            print(f"[WARN][TRAIN] Batch {batch_idx}: entrada no
finita detectada. Se omite.")
            continue

        valid = (displ > 0) & (displ < maxdispl)

        if valid.sum().item() == 0:
            skipped_batches_invalid_gt += 1
            continue

        optimizer.zero_grad()

        pred1, pred2, pred3 = model(imgL, imgR)

        pred1 = pred1.squeeze(1) if pred1.ndim == 4 else pred1
        pred2 = pred2.squeeze(1) if pred2.ndim == 4 else pred2
        pred3 = pred3.squeeze(1) if pred3.ndim == 4 else pred3

        if not torch.isfinite(pred1).all() or not
torch.isfinite(pred2).all() or not torch.isfinite(pred3).all():
            skipped_batches_nonfinite += 1
            print(f"[WARN][TRAIN] Batch {batch_idx}: predicción no
finita detectada. Se omite.")
            optimizer.zero_grad()
            continue

        loss = (
            0.5 * F.smooth_l1_loss(pred1[valid], displ[valid],
reduction="mean") +
            0.7 * F.smooth_l1_loss(pred2[valid], displ[valid],
reduction="mean") +
            1.0 * F.smooth_l1_loss(pred3[valid], displ[valid],
reduction="mean")
        )

        if not torch.isfinite(loss):
            skipped_batches_nonfinite += 1

```

```

        print(f"[WARN][TRAIN] Batch {batch_idx}: loss no finita
detectada. Se omite.")
        optimizer.zero_grad()
        continue

    loss.backward()

    grads_are_finite = True

    for p in model.parameters():
        if p.grad is not None and not
torch.isfinite(p.grad).all():
            grads_are_finite = False
            break

    if not grads_are_finite:
        skipped_batches_nonfinite += 1
        print(f"[WARN][TRAIN] Batch {batch_idx}: gradientes no
finitos detectados. Se omite actualización.")
        optimizer.zero_grad()
        continue

    optimizer.step()
    losses.append(loss.item())

    print(f"[TRAIN] Batches omitidos sin GT válido:
{skipped_batches_invalid_gt}")
    print(f"[TRAIN] Batches omitidos por NaN/Inf:
{skipped_batches_nonfinite}")

    if len(losses) == 0:
        raise RuntimeError("No hubo ningún batch válido de
entrenamiento en la época.")

    epoch_loss = float(np.mean(losses))

    if not np.isfinite(epoch_loss):
        raise RuntimeError("La loss media de la época no es
finita.")

```

```

    return epoch_loss

@torch.no_grad()
def evaluate_selection(model, loader, device, maxdisp=192):
    model.eval()

    dl_list = []
    epe_list = []
    abs_rel_list = []
    delta1_list = []

    skipped_samples_nonfinite = 0
    skipped_samples_invalid = 0

    for sample_idx, (imgL, imgR, dispL) in enumerate(loader):
        imgL = imgL.float().to(device, non_blocking=True)
        imgR = imgR.float().to(device, non_blocking=True)
        dispL = dispL.float().to(device, non_blocking=True)

        if not torch.isfinite(imgL).all() or not
torch.isfinite(imgR).all() or not torch.isfinite(dispL).all():
            skipped_samples_nonfinite += 1
            print(f"[WARN][EVAL] Muestra {sample_idx}: entrada no
finita detectada. Se omite.")
            continue

        pred = model(imgL, imgR)

        if isinstance(pred, (tuple, list)):
            pred = pred[-1]

        pred = pred.squeeze(1) if pred.ndim == 4 else pred

        if not torch.isfinite(pred).all():
            skipped_samples_nonfinite += 1
            print(f"[WARN][EVAL] Muestra {sample_idx}: predicción
no finita detectada. Se omite.")
            continue

```



```

        m = compute_metrics_from_prediction(pred, dispL,
maxdisp=maxdisp)

        if (
            not np.isfinite(m["d1_all"])
            or not np.isfinite(m["epe"])
            or not np.isfinite(m["abs_rel"])
            or not np.isfinite(m["delta1"])
        ):
            skipped_samples_invalid += 1
            print(f"[WARN][EVAL] Muestra {sample_idx}: métricas
inválidas. Se omite.")
            continue

            d1_list.append(m["d1_all"])
            epe_list.append(m["epe"])
            abs_rel_list.append(m["abs_rel"])
            delta1_list.append(m["delta1"])

        print(f"[EVAL] Muestras omitidas por NaN/Inf:
{skipped_samples_nonfinite}")
        print(f"[EVAL] Muestras omitidas por métricas inválidas:
{skipped_samples_invalid}")

        if (
            len(d1_list) == 0
            or len(epe_list) == 0
            or len(abs_rel_list) == 0
            or len(delta1_list) == 0
        ):
            raise RuntimeError("La evaluación no produjo métricas
válidas en esta época.")

        d1_mean = float(np.mean(d1_list))
        epe_mean = float(np.mean(epe_list))
        abs_rel_mean = float(np.mean(abs_rel_list))
        delta1_mean = float(np.mean(delta1_list))

        if (
            not np.isfinite(d1_mean)

```

```

        or not np.isfinite(epe_mean)
        or not np.isfinite(abs_rel_mean)
        or not np.isfinite(delta1_mean)
    ):
        raise RuntimeError("Las métricas medias de evaluación no
son finitas.")

    return {
        "dl_all": dl_mean,
        "epe": epe_mean,
        "abs_rel": abs_rel_mean,
        "delta1": delta1_mean,
    }

```

GUARDAR PESOS

```

def save_epoch_weights(model, optimizer, epoch, out_dir,
    history_row=None):
    ensure_dir(out_dir)

    filename = f"pesos_epoca{epoch:02d}.tar"
    out_path = os.path.join(out_dir, filename)

    payload = {
        "epoch": epoch,
        "state_dict": model.state_dict(),
        "optimizer_state_dict": optimizer.state_dict(),
    }

    if history_row is not None:
        payload["metrics"] = history_row

    torch.save(payload, out_path)

    if not os.path.exists(out_path) or os.path.getsize(out_path) ==
0:
        raise RuntimeError(f"No se pudo guardar correctamente el
checkpoint de la época {epoch}.")

```

```

    return out_path

def save_last_epoch_weights(model, optimizer, epoch, out_dir,
    history_row=None):
    ensure_dir(out_dir)

    filename = f"pesos_ultima_epoca{epoch:02d}.tar"
    out_path = os.path.join(out_dir, filename)

    payload = {
        "epoch": epoch,
        "state_dict": model.state_dict(),
        "optimizer_state_dict": optimizer.state_dict(),
    }

    if history_row is not None:
        payload["metrics"] = history_row

    torch.save(payload, out_path)

    if not os.path.exists(out_path) or os.path.getsize(out_path) ==
0:
        raise RuntimeError(f"No se pudo guardar correctamente el
checkpoint final de la época {epoch}.")

    return out_path

# FUNCIÓN PRINCIPAL

def run_finetuning_experiment(
    trainable_mode,
    drive_output_dir,
    config,
    prepared_data=None
):

    assert_gpu_available()
    set_all_seeds(config["seed"])

```

```

ensure_dir(drive_output_dir)

if prepared_data is None:
    print("No se recibió prepared_data. Preparando datos dentro
de la función...")
    prepared_data = prepare_all_data(config)

required_prepared_keys = {"kitti2012_root", "kitti2015_root",
"pretrained_path"}

if not isinstance(prepared_data, dict) or not
required_prepared_keys.issubset(set(prepared_data.keys())):
    raise RuntimeError("prepared_data no es válido o le faltan
claves requeridas.")

pesos_por_epoca_dir = os.path.join(drive_output_dir,
"PesosPorEpoca")
ultima_epoca_dir = os.path.join(drive_output_dir,
"UltimaEpoca")
ensure_dir(pesos_por_epoca_dir)
ensure_dir(ultima_epoca_dir)

device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

print("=====
===")
print("INICIO DEL EXPERIMENTO")

print("=====
===")
print("Device:", device)
print("Modo de entrenamiento:", trainable_mode)
print("Salida en Drive:", drive_output_dir)
print("Número total de épocas:", config["max_epochs"])

print("=====
===")

```

```

train_loader, selection_loader, train_names, selection_names =
make_loaders(
    kitti2012_root=prepared_data["kitti2012_root"],
    kitti2015_root=prepared_data["kitti2015_root"],
    batch_size_train=config["batch_size_train"],
    batch_size_eval=config["batch_size_eval"],
    num_workers=config["num_workers"],
)

model = stackhourglass(config["maxdisp"]).to(device)
model = load_official_checkpoint(model,
prepared_data["pretrained_path"], device)

layer_info = configure_trainable_layers(model, trainable_mode)

print(f"Parámetros entrenables:
{count_trainable_params(model):,}")

optimizer = optim.Adam(
    [p for p in model.parameters() if p.requires_grad],
    lr=config["lr"],
    betas=(0.9, 0.999)
)

save_list_txt(os.path.join(drive_output_dir,
"kitti2012_train_samples.txt"), train_names)
save_list_txt(os.path.join(drive_output_dir,
"kitti2015_selection_samples.txt"), selection_names)

with open(os.path.join(drive_output_dir,
"layer_mode_info.json"), "w") as f:
    json.dump(layer_info, f, indent=2)

experiment_info = {
    "trainable_mode": trainable_mode,
    "seed": config["seed"],
    "maxdisp": config["maxdisp"],
    "batch_size_train": config["batch_size_train"],
    "batch_size_eval": config["batch_size_eval"],
    "lr": config["lr"],

```

```

    "min_delta": config["min_delta"],
    "max_epochs": config["max_epochs"],
    "kitti2012_root": prepared_data["kitti2012_root"],
    "kitti2015_root": prepared_data["kitti2015_root"],
    "pretrained_path": prepared_data["pretrained_path"],
}

with open(os.path.join(drive_output_dir,
"experiment_config.json"), "w") as f:
    json.dump(experiment_info, f, indent=2)

history = []
best_epoch = None
best_d1 = float("inf")
last_epoch_weights_path = None

metrics_csv_path = os.path.join(drive_output_dir,
"metrics_history.csv")
summary_txt_path = os.path.join(drive_output_dir,
"summary.txt")

total_start = time.time()

for epoch in range(1, config["max_epochs"] + 1):
    epoch_start = time.time()

print("\n-----")
print(f"ÉPOCA ACTUAL: {epoch}/{config['max_epochs']}")

print("-----")

train_loss = train_one_epoch(
    model=model,
    loader=train_loader,
    optimizer=optimizer,
    device=device,
    maxdisp=config["maxdisp"],

```

```

    )

    sel_metrics = evaluate_selection(
        model=model,
        loader=selection_loader,
        device=device,
        maxdisp=config["maxdisp"],
    )

    if not np.isfinite(train_loss):
        raise RuntimeError(f"train_loss no finita en la época
{epoch}.")

    if (
        not np.isfinite(sel_metrics["d1_all"])
        or not np.isfinite(sel_metrics["epe"])
        or not np.isfinite(sel_metrics["abs_rel"])
        or not np.isfinite(sel_metrics["delta1"])
    ):
        raise RuntimeError(f"Métricas de selección no finitas
en la época {epoch}.")

    row = {
        "epoch": epoch,
        "train_loss": train_loss,
        "d1_all": sel_metrics["d1_all"],
        "epe": sel_metrics["epe"],
        "abs_rel": sel_metrics["abs_rel"],
        "delta1": sel_metrics["delta1"],
    }

    history.append(row)
    save_history_as_csv(history, metrics_csv_path)

    improved = (best_d1 - sel_metrics["d1_all"]) >
config["min_delta"]

    print(
        f"Resultado época {epoch:02d}: "
        f"train_loss={train_loss:.6f} | "

```

```

        f"d1_all={sel_metrics['d1_all']:.6f} | "
        f"epe={sel_metrics['epe']:.6f} | "
        f"abs_rel={sel_metrics['abs_rel']:.6f} | "
        f"delta1={sel_metrics['delta1']:.6f}"
    )

    if improved:
        epoch_weights_path = save_epoch_weights(
            model=model,
            optimizer=optimizer,
            epoch=epoch,
            out_dir=pesos_por_epoca_dir,
            history_row=row
        )

        best_d1 = sel_metrics["d1_all"]
        best_epoch = epoch

        print(f"Mejora detectada en D1-all. Mejor época provisional: {best_epoch}")
        print(f"Pesos guardados en: {epoch_weights_path}")
    else:
        print("Sin mejora en D1-all. No se guardan pesos de esta época como mejor modelo.")

    if epoch == config["max_epochs"]:
        last_epoch_weights_path = save_last_epoch_weights(
            model=model,
            optimizer=optimizer,
            epoch=epoch,
            out_dir=ultima_epoca_dir,
            history_row=row
        )
        print(f"Pesos de la última época guardados en: {last_epoch_weights_path}")

    epoch_time = time.time() - epoch_start
    print(f"Tiempo de la época: {epoch_time:.2f} s")

```



```

total_time = time.time() - total_start

with open(summary_txt_path, "w") as f:
    f.write("RESUMEN FINAL\n")
    f.write(f"trainable_mode\t{trainable_mode}\n")
    f.write(f"max_epochs_config\t{config['max_epochs']}\n")
    f.write(f"executed_epochs\t{len(history)}\n")
    f.write(f"best_epoch_by_d1_all\t{best_epoch}\n")
    f.write(f"best_d1_all\t{best_d1:.6f}\n")
    f.write(f"total_time_sec\t{total_time:.2f}\n")
    f.write(f"best_weights_folder\t{pesos_por_epoca_dir}\n")

f.write(f"last_epoch_weights_file\t{last_epoch_weights_path}\n")
f.write(f"metrics_file\t{metrics_csv_path}\n")

print("\n=====
=====")
print("FIN DEL EXPERIMENTO")

print("=====
=====")
print(f"Épocas ejecutadas: {len(history)}")
print(f"Mejor época provisional según D1-all: {best_epoch}")
print(f"Mejor D1-all observado: {best_d1:.6f}")
print(f"Histórico guardado en: {metrics_csv_path}")
print(f"Resumen guardado en: {summary_txt_path}")
print(f"Pesos con nuevo mínimo de D1-all guardados en:
{pesos_por_epoca_dir}")
print(f"Pesos de la última época guardados en:
{last_epoch_weights_path}")

print("=====
=====")

return {
    "executed_epochs": len(history),
    "best_epoch_by_d1_all": best_epoch,
    "best_d1_all": best_d1,
    "metrics_csv_path": metrics_csv_path,

```

```

        "summary_txt_path": summary_txt_path,
        "best_weights_dir": pesos_por_epoca_dir,
        "last_epoch_weights_path": last_epoch_weights_path,
    }

# PREPARAR DATOS

prepared = prepare_all_data(CONFIG)
prepared

# EJECUCIÓN

# Lógica preparada para los tres modos:
#
# resultado_todas = run_finetuning_experiment(
#     trainable_mode="todas_capas",
#
#     drive_output_dir="/content/drive/MyDrive/TFG/Entrenamiento/Todas
#     Capas",
#     config=CONFIG,
#     prepared_data=prepared
# )
#
# resultado_intermedias_finales = run_finetuning_experiment(
#     trainable_mode="intermedias_finales",
#
#     drive_output_dir="/content/drive/MyDrive/TFG/Entrenamiento/Media
#     syfinalesCapas",
#     config=CONFIG,
#     prepared_data=prepared
# )
#

```

```

# resultado_solo_finales = run_finetuning_experiment(
#     trainable_mode="solo_finales",
#     drive_output_dir="/content/drive/MyDrive/TFG/Entrenamiento/Solof
#         inalesCapas",
#     config=CONFIG,
#     prepared_data=prepared
# )

#Entrenamiento de todas las capas, sustituir para otro experimento:
resultado_todas = run_finetuning_experiment(
    trainable_mode="todas_capas",

    drive_output_dir="/content/drive/MyDrive/TFG/Entrenamiento/Todas
    Capas",
    config=CONFIG,
    prepared_data=prepared
)

```

```
runtime.unassign()
```



7.3. ANEXO 3: EVALUACIÓN DEL FINE-TUNED Y PREENTRENO

```
# =====
# EVALUACIÓN FINAL PSMNet - KITTI2015 SEGUNDA MITAD
# Evalúa:
#   - CNN preentrenada
#   - TodasCapas: MejorDlTeorica y UltimaEpoca
#   - Mediasy finalesCapas: MejorDlTeorica y UltimaEpoca
#   - SolofinalesCapas: MejorDlTeorica y UltimaEpoca
#
# Guarda:
#   - metrics_per_image.csv
#   - metrics_summary.csv
#   - metricas_añadidas.csv
#   - metricas_añadidas_resumen.csv
#   - evaluation_config.json
#
# Métricas base:
#   - D1-all
#   - EPE
#   - AbsRel
#   - delta1
#
# Métricas añadidas:
#   - D1-all, EPE, AbsRel y delta1 por distancia:
#       cerca: 0-30 m
#       media: 30-60 m
#       lejos: 60-90 m
# =====

import os
import sys
import csv
import json
import time
import shutil
import zipfile
from pathlib import Path

import numpy as np
```

```

import torch
from torch.utils.data import DataLoader

from google.colab import drive
from google.colab import runtime

# =====
# 1. CONFIGURACIÓN
# =====

drive.mount("/content/drive", force_remount=True)

CONFIG = {
    "kitti2015_zip_drive":
"/content/drive/MyDrive/TFG/Datasets/KITTI2015/data_scene_flow.zip"
,
    "kitti2015_calib_zip_drive":
"/content/drive/MyDrive/TFG/Datasets/KITTI2015/data_scene_flow_calib.zip",

    "psmnet_repo_drive":
"/content/drive/MyDrive/TFG/PSMNet-master",

    "maxdisp": 192,
    "batch_size_eval": 1,
    "num_workers": 2,

    # División para métricas por distancia:
    # cerca: 0-30 m, media: >30-60 m, lejos: >60-90 m
    "max_depth_m": 90.0,

    "output_base_dir": "/content/drive/MyDrive/TFG/Evaluacion",
}

```

```

EXPERIMENTOS = {
    "preentrenado": {
        "weights": {
            "preentrenado":
"/content/drive/MyDrive/TFG/Evaluacion/CnnPreentrenada/Pesos/pretra
ined_sceneflow_new.tar",
        },
        "results_dir":
"/content/drive/MyDrive/TFG/Evaluacion/CnnPreentrenada/Resultados",
    },

    "todas_capas": {
        "weights": {
            "mejor_dl":
"/content/drive/MyDrive/TFG/Evaluacion/TodasCapas/Pesos/pesos_mejor
_dlteorica.tar",
            "ultima_epoca":
"/content/drive/MyDrive/TFG/Evaluacion/TodasCapas/Pesos/pesos_ultim
a_epoca300.tar",
        },
        "results_dir":
"/content/drive/MyDrive/TFG/Evaluacion/TodasCapas/Resultados",
    },

    "intermedias_finales": {
        "weights": {
            "mejor_dl":
"/content/drive/MyDrive/TFG/Evaluacion/MediasyfinalesCapas/Pesos/pe
sos_mejor_dlteorica.tar",
            "ultima_epoca":
"/content/drive/MyDrive/TFG/Evaluacion/MediasyfinalesCapas/Pesos/pe
sos_ultima_epoca300.tar",
        },
        "results_dir":
"/content/drive/MyDrive/TFG/Evaluacion/MediasyfinalesCapas/Resultad
os",
    },

    "solo_finales": {
        "weights": {

```

```

        "mejor_dl":
"/content/drive/MyDrive/TFG/Evaluacion/SolofinalesCapas/Pesos/pesos
_mejor_dlteorica.tar",
        "ultima_epoca":
"/content/drive/MyDrive/TFG/Evaluacion/SolofinalesCapas/Pesos/pesos
_ultima_epoca300.tar",
    },
    "results_dir":
"/content/drive/MyDrive/TFG/Evaluacion/SolofinalesCapas/Resultados"
,
    },
}

```

```

# =====
# 2. UTILIDADES GENERALES
# =====

```

```

def ensure_dir(path):
    Path(path).mkdir(parents=True, exist_ok=True)

def remove_dir_if_exists(path):
    p = Path(path)
    if p.exists():
        shutil.rmtree(p)

def assert_gpu_available():
    if not torch.cuda.is_available():
        raise RuntimeError(
            "No hay GPU activa. En Colab ve a Entorno de ejecución
> Cambiar tipo de entorno de ejecución > GPU."
        )

def copy_zip_from_drive_to_content(src_drive_path,
dst_dir="/content/datasets_zip"):
    ensure_dir(dst_dir)

```

```

src = Path(src_drive_path)

if not src.exists():
    raise FileNotFoundError(f"No existe el ZIP en Drive:
{src_drive_path}")

dst = Path(dst_dir) / src.name

print(f"Copiando ZIP desde Drive:")
print(f" Origen: {src}")
print(f" Destino: {dst}")

shutil.copy2(src, dst)

if not dst.exists() or dst.stat().st_size == 0:
    raise RuntimeError(f"Error al copiar ZIP: {dst}")

return str(dst)

def unzip_file(zip_path, extract_to, clean_before_extract=True):
    if clean_before_extract:
        remove_dir_if_exists(extract_to)

    ensure_dir(extract_to)

    print(f"Descomprimiendo:")
    print(f" ZIP: {zip_path}")
    print(f" Destino: {extract_to}")

    try:
        with zipfile.ZipFile(zip_path, "r") as zf:
            bad_member = zf.testzip()

            if bad_member is not None:
                raise RuntimeError(
                    f"ZIP corrupto o incompleto. Primer archivo
problemático: {bad_member}"
                )

```



```

        zf.extractall(extract_to)

    except zipfile.BadZipFile:
        raise RuntimeError(f"Archivo ZIP no válido: {zip_path}")

    return extract_to


def find_kitti2015_training_root(search_root):
    required = {"image_2", "image_3", "disp_occ_0"}

    for root, dirs, files in os.walk(search_root):
        if required.issubset(set(dirs)):
            return root + "/"

    raise FileNotFoundError(
        f"No se encontró la raíz training de KITTI2015 dentro de {search_root}"
    )


def find_kitti2015_calib_root(search_root):
    search_root = Path(search_root)

    preferred = search_root / "training" / "calib_cam_to_cam"

    if preferred.exists():
        txt_files = list(preferred.glob("*.txt"))
        numeric_txt = [f for f in txt_files if len(f.name) >= 10
            and f.name[:6].isdigit()]

        if len(numeric_txt) > 0:
            return str(preferred) + "/"

    candidates = []

    for root, dirs, files in os.walk(search_root):
        root_path = Path(root)

        if "training" not in root_path.parts:

```

```

        continue

    txt_files = [f for f in files if f.endswith(".txt")]
    numeric_txt = [f for f in txt_files if len(f) >= 10 and
f[:6].isdigit()]

    if len(numeric_txt) > 0:
        candidates.append(root)

if not candidates:
    raise FileNotFoundError(
        f"No se encontró carpeta de calibración de training
dentro de {search_root}"
    )

candidates = sorted(candidates)

return candidates[0] + "/"

def prepare_kitti2015(config):
    local_zip_dir = "/content/datasets_zip"

    kitti_zip = copy_zip_from_drive_to_content(
        config["kitti2015_zip_drive"],
        local_zip_dir
    )

    kitti_extract = unzip_file(
        kitti_zip,
        "/content/datasets/kitti2015",
        clean_before_extract=True
    )

    kitti_root = find_kitti2015_training_root(kitti_extract)

    calib_zip = copy_zip_from_drive_to_content(
        config["kitti2015_calib_zip_drive"],
        local_zip_dir
    )

```

```

calib_extract = unzip_file(
    calib_zip,
    "/content/datasets/kitti2015_calib",
    clean_before_extract=True
)

calib_root = find_kitti2015_calib_root(calib_extract)

print("KITTI2015 preparado correctamente.")
print(f"Raíz imágenes/GT: {kitti_root}")
print(f"Raíz calibración: {calib_root}")

return kitti_root, calib_root

# =====
# 3. IMPORTAR PSMNet DESDE DRIVE
# =====

def import_psmnet_from_drive(repo_drive_path):
    repo_dir = Path(repo_drive_path)

    if not repo_dir.exists():
        raise FileNotFoundError(
            f"No existe el repositorio PSMNet en Drive: {repo_drive_path}"
        )

    expected_files = [
        repo_dir / "models" / "__init__.py",
        repo_dir / "dataloader" / "KITTIloader.py",
        repo_dir / "dataloader" / "preprocess.py",
    ]

    for f in expected_files:
        if not f.exists():
            raise FileNotFoundError(f"Falta archivo esperado del repo PSMNet: {f}")

```

```

repo_dir_str = str(repo_dir.resolve())
dataloader_dir_str = str((repo_dir / "dataloader").resolve())
models_dir_str = str((repo_dir / "models").resolve())

for p in [repo_dir_str, dataloader_dir_str, models_dir_str]:
    if p not in sys.path:
        sys.path.insert(0, p)

from models import stackhourglass
from dataloader.KITTIloader import myImageFloder

print(f"PSMNet importado correctamente desde Drive:
{repo_drive_path}")

return stackhourglass, myImageFloder

# =====
# 4. LISTAS KITTI2015 SEGUNDA MITAD
# =====

def build_kitti2015_final_eval_lists_second_half(kitti2015_root,
calib_root):
    left_dir = Path(kitti2015_root) / "image_2"
    right_dir = Path(kitti2015_root) / "image_3"
    disp_dir = Path(kitti2015_root) / "disp_occ_0"
    calib_dir = Path(calib_root)

    requested = []

    for i in range(100, 200):
        requested.append(f"{i:06d}_10.png")
        requested.append(f"{i:06d}_11.png")

    left_names = {p.name for p in left_dir.glob("*.png")}
    right_names = {p.name for p in right_dir.glob("*.png")}
    disp_names = {p.name for p in disp_dir.glob("*.png")}
    calib_names = {p.name for p in calib_dir.glob("*.txt")}

    common = []

```

```

missing = []

for n in requested:
    base_id = n[:6]
    calib_name = f"{base_id}.txt"

    ok = (
        n in left_names
        and n in right_names
        and n in disp_names
        and calib_name in calib_names
    )

    if ok:
        common.append(n)
    else:
        missing.append(n)

if len(common) == 0:
    raise RuntimeError(
        "No se encontraron muestras válidas en la segunda mitad
de KITTI2015."
    )

if len(missing) > 0:
    print(f"[WARN] Hay {len(missing)} muestras solicitadas que
no se encontraron completas.")
    print("Primeras muestras omitidas:", missing[:10])

left = [str(left_dir / n) for n in common]
right = [str(right_dir / n) for n in common]
disp = [str(disp_dir / n) for n in common]
calib = [str(calib_dir / f"{n[:6]}.txt") for n in common]

print(f"KITTI2015 evaluación final: {len(common)} muestras
válidas.")

return left, right, disp, calib, common

```

```

def make_eval_loader(myImageFloder, kitti2015_root, calib_root,
config):
    left, right, disp, calib, names =
build_kitti2015_final_eval_lists_second_half(
    kitti2015_root=kitti2015_root,
    calib_root=calib_root,
)

dataset = myImageFloder(left, right, disp, training=False)

loader = DataLoader(
    dataset,
    batch_size=config["batch_size_eval"],
    shuffle=False,
    num_workers=config["num_workers"],
    drop_last=False,
    pin_memory=True,
)

return loader, names, calib

# =====
# 5. CALIBRACIÓN POR IMAGEN
# =====

def parse_kitti_calib_file(calib_path):
    data = {}

    with open(calib_path, "r") as f:
        for line in f:
            if ":" not in line:
                continue

            key, value = line.split(":", 1)
            key = key.strip()
            value = value.strip()

            try:
                values = [float(x) for x in value.split()]

```

```

        except ValueError:
            continue

        data[key] = values

    return data

def get_fx_baseline_from_calib(calib_path):
    data = parse_kitti_calib_file(calib_path)

    if "P2" in data and "P3" in data:
        P2_key = "P2"
        P3_key = "P3"
    elif "P_rect_02" in data and "P_rect_03" in data:
        P2_key = "P_rect_02"
        P3_key = "P_rect_03"
    else:
        available_keys = sorted(data.keys())
        raise KeyError(
            f"No se encontraron P2/P3 ni P_rect_02/P_rect_03 en "
            f"calibración: {calib_path}. "
            f"Claves disponibles: {available_keys}"
        )

    P2 = np.array(data[P2_key], dtype=np.float64).reshape(3, 4)
    P3 = np.array(data[P3_key], dtype=np.float64).reshape(3, 4)

    fx = P2[0, 0]

    tx2 = -P2[0, 3] / P2[0, 0]
    tx3 = -P3[0, 3] / P3[0, 0]

    baseline = abs(tx3 - tx2)

    if fx <= 0 or baseline <= 0:
        raise RuntimeError(
            f"Calibración inválida en {calib_path}. fx={fx}, "
            f"baseline={baseline}"
        )

```

```

    return float(fx), float(baseline)

# =====
# 6. CARGA DE CHECKPOINTS
# =====

def load_checkpoint_into_model(model, ckpt_path, device):
    if not os.path.exists(ckpt_path):
        raise FileNotFoundError(f"No existe el checkpoint: {ckpt_path}")

    if os.path.getsize(ckpt_path) == 0:
        raise RuntimeError(f"El checkpoint está vacío: {ckpt_path}")

    ckpt = torch.load(ckpt_path, map_location=device)

    if isinstance(ckpt, dict) and "state_dict" in ckpt:
        state = ckpt["state_dict"]
    else:
        state = ckpt

    if not isinstance(state, dict):
        raise RuntimeError(
            f"El checkpoint no contiene un state_dict válido: {ckpt_path}"
        )

    clean_state = {}

    for k, v in state.items():
        clean_state[k.replace("module.", "")] = v

    model_state = model.state_dict()

    missing_keys = sorted(set(model_state.keys()) -
                           set(clean_state.keys()))

```



```

unexpected_keys = sorted(set(clean_state.keys()) -
set(model_state.keys()))

shape_mismatch = []

for k in set(model_state.keys()) & set(clean_state.keys()):
    if model_state[k].shape != clean_state[k].shape:
        shape_mismatch.append(
            (k, tuple(clean_state[k].shape),
tuple(model_state[k].shape))

        )

    if missing_keys:
        msg = "\n".join(missing_keys[:30])
        raise RuntimeError(
            f"Faltan claves del checkpoint al cargar el modelo.
Primeras:\n{msg}"
        )

    if unexpected_keys:
        msg = "\n".join(unexpected_keys[:30])
        raise RuntimeError(
            f"Hay claves inesperadas en el checkpoint.
Primeras:\n{msg}"
        )

    if shape_mismatch:
        preview = "\n".join(
            [f"{k}: ckpt{a} vs model{b}" for k, a, b in
shape_mismatch[:30]]
        )
        raise RuntimeError(
            f"Hay tensores con forma incompatible en el checkpoint.
Primeros:\n{preview}"
        )

model.load_state_dict(clean_state, strict=True)

print(f"Checkpoint cargado correctamente:")
print(f" {ckpt_path}")

```

```

    return model

# =====
# 7. MÉTRICAS BASE
# =====

def compute_basic_metrics(pred_disp, gt_disp, maxdisp=192):
    valid = (gt_disp > 0) & (gt_disp < maxdisp)

    valid_pixels = int(valid.sum().item())

    if valid_pixels == 0:
        return {
            "d1_all": np.nan,
            "epe": np.nan,
            "abs_rel": np.nan,
            "delta1": np.nan,
            "valid_pixels": 0,
        }

    pred_valid = pred_disp[valid]
    gt_valid = gt_disp[valid]

    abs_err = torch.abs(pred_valid - gt_valid)

    if abs_err.numel() == 0 or not torch.isfinite(abs_err).all():
        return {
            "d1_all": np.nan,
            "epe": np.nan,
            "abs_rel": np.nan,
            "delta1": np.nan,
            "valid_pixels": valid_pixels,
        }

    epe = abs_err.mean().item()

    d1_all = (
        ((abs_err > 3.0) & (abs_err > 0.05 * gt_valid))

```

```

        .float()
        .mean()
        .item()
    )

    abs_rel = (
        abs_err / torch.clamp(gt_valid, min=1e-6)
    ).mean().item()

    pred_safe = torch.clamp(pred_valid, min=1e-6)
    gt_safe = torch.clamp(gt_valid, min=1e-6)

    ratio = torch.maximum(pred_safe / gt_safe, gt_safe / pred_safe)
    delta1 = (ratio < 1.25).float().mean().item()

    return {
        "d1_all": d1_all,
        "epe": epe,
        "abs_rel": abs_rel,
        "delta1": delta1,
        "valid_pixels": valid_pixels,
    }

def compute_metrics_on_mask(pred_disp, gt_disp, mask):
    num_pixels = int(mask.sum().item())

    if num_pixels == 0:
        return {
            "d1_all": np.nan,
            "epe": np.nan,
            "abs_rel": np.nan,
            "delta1": np.nan,
            "num_pixels": 0,
        }

    pred_valid = pred_disp[mask]
    gt_valid = gt_disp[mask]

    abs_err = torch.abs(pred_valid - gt_valid)

```

```

if abs_err.numel() == 0 or not torch.isfinite(abs_err).all():
    return {
        "d1_all": np.nan,
        "epe": np.nan,
        "abs_rel": np.nan,
        "delta1": np.nan,
        "num_pixels": num_pixels,
    }

epe = abs_err.mean().item()

d1_all = (
    ((abs_err > 3.0) & (abs_err > 0.05 * gt_valid))
    .float()
    .mean()
    .item()
)

abs_rel = (
    abs_err / torch.clamp(gt_valid, min=1e-6)
).mean().item()

pred_safe = torch.clamp(pred_valid, min=1e-6)
gt_safe = torch.clamp(gt_valid, min=1e-6)

ratio = torch.maximum(pred_safe / gt_safe, gt_safe / pred_safe)
delta1 = (ratio < 1.25).float().mean().item()

return {
    "d1_all": d1_all,
    "epe": epe,
    "abs_rel": abs_rel,
    "delta1": delta1,
    "num_pixels": num_pixels,
}

```

```

# =====
# 8. MÉTRICAS AÑADIDAS: DISTANCIA
# =====

def disparity_to_depth_m(disparity, fx, baseline):
    return (fx * baseline) / torch.clamp(disparity, min=1e-6)

def make_depth_masks(gt_disp, valid_mask, fx, baseline,
max_depth_m):
    gt_depth = disparity_to_depth_m(gt_disp, fx, baseline)

    close_max = max_depth_m / 3.0
    medium_max = 2.0 * max_depth_m / 3.0

    masks = {
        "cerca": valid_mask & (gt_depth > 0.0) & (gt_depth <=
close_max),
        "media": valid_mask & (gt_depth > close_max) & (gt_depth <=
medium_max),
        "lejos": valid_mask & (gt_depth > medium_max) & (gt_depth
<= max_depth_m),
    }

    ranges = {
        "cerca": f"0-{close_max:.1f} m",
        "media": f">{close_max:.1f}-{medium_max:.1f} m",
        "lejos": f">{medium_max:.1f}-{max_depth_m:.1f} m",
    }

    return masks, ranges

def compute_added_metrics_rows(pred_disp, gt_disp, sample_name,
calib_path, config):
    fx, baseline = get_fx_baseline_from_calib(calib_path)

    valid = (gt_disp > 0) & (gt_disp < config["maxdisp"])

    rows = []

```

```

depth_masks, depth_ranges = make_depth_masks(
    gt_disp=gt_disp,
    valid_mask=valid,
    fx=fx,
    baseline=baseline,
    max_depth_m=config["max_depth_m"],
)

for zone_name, mask in depth_masks.items():
    m = compute_metrics_on_mask(pred_disp, gt_disp, mask)

    rows.append({
        "sample": sample_name,
        "group_type": "distancia",
        "group_name": zone_name,
        "range_m": depth_ranges[zone_name],
        "fx": fx,
        "baseline": baseline,
        "d1_all": m["d1_all"],
        "epe": m["epe"],
        "abs_rel": m["abs_rel"],
        "delta1": m["delta1"],
        "num_pixels": m["num_pixels"],
    })

return rows

# =====
# 9. GUARDADO CSV
# =====

def safe_float(x):
    if x is None:
        return "nan"

    if isinstance(x, (float, np.floating)):
        if not np.isfinite(x):
            return "nan"

```

```

        return f"{float(x):.6f}"

    return x

def save_metrics_per_image(rows, out_path):
    with open(out_path, "w", newline="") as f:
        writer = csv.writer(f)

        writer.writerow([
            "sample",
            "d1_all",
            "epe",
            "abs_rel",
            "delta1",
            "valid_pixels",
        ])

        for r in rows:
            writer.writerow([
                r["sample"],
                safe_float(r["d1_all"]),
                safe_float(r["epe"]),
                safe_float(r["abs_rel"]),
                safe_float(r["delta1"]),
                r["valid_pixels"],
            ])

def save_metrics_summary(rows, out_path):
    metrics = ["d1_all", "epe", "abs_rel", "delta1"]

    with open(out_path, "w", newline="") as f:
        writer = csv.writer(f)

        writer.writerow(["metric", "value"])
        writer.writerow(["num_samples", len(rows)])
        writer.writerow(["total_valid_pixels",
sum(r["valid_pixels"] for r in rows)])

```

```

for m in metrics:
    vals = [r[m] for r in rows if np.isfinite(r[m])]
    mean_val = float(np.mean(vals)) if vals else np.nan
    writer.writerow([m, safe_float(mean_val)])

def save_added_metrics(rows, out_path):
    with open(out_path, "w", newline="") as f:
        writer = csv.writer(f)

        writer.writerow([
            "sample",
            "group_type",
            "group_name",
            "range_m",
            "fx",
            "baseline",
            "d1_all",
            "epe",
            "abs_rel",
            "delta1",
            "num_pixels",
        ])

    for r in rows:
        writer.writerow([
            r["sample"],
            r["group_type"],
            r["group_name"],
            r["range_m"],
            safe_float(r["fx"]),
            safe_float(r["baseline"]),
            safe_float(r["d1_all"]),
            safe_float(r["epe"]),
            safe_float(r["abs_rel"]),
            safe_float(r["delta1"]),
            r["num_pixels"],
        ])

```



```

def save_added_metrics_summary(rows, out_path):
    grouped = {}

    for r in rows:
        key = (r["group_type"], r["group_name"], r["range_m"])

        if key not in grouped:
            grouped[key] = {
                "d1_all": [],
                "epe": [],
                "abs_rel": [],
                "deltal": [],
                "num_pixels": 0,
            }

        for metric in ["d1_all", "epe", "abs_rel", "deltal"]:
            if np.isfinite(r[metric]):
                grouped[key][metric].append(r[metric])

        grouped[key]["num_pixels"] += r["num_pixels"]

    with open(out_path, "w", newline="") as f:
        writer = csv.writer(f)

        writer.writerow([
            "group_type",
            "group_name",
            "range_m",
            "d1_all_mean",
            "epe_mean",
            "abs_rel_mean",
            "deltal_mean",
            "total_pixels",
        ])

    for key, values in grouped.items():
        group_type, group_name, range_m = key

        d1_mean = (
            float(np.mean(values["d1_all"]))

```

```

        if values["dl_all"]
        else np.nan
    )

    epe_mean = (
        float(np.mean(values["epe"]))
        if values["epe"]
        else np.nan
    )

    abs_rel_mean = (
        float(np.mean(values["abs_rel"]))
        if values["abs_rel"]
        else np.nan
    )

    delta1_mean = (
        float(np.mean(values["delta1"]))
        if values["delta1"]
        else np.nan
    )

    writer.writerow([
        group_type,
        group_name,
        range_m,
        safe_float(dl_mean),
        safe_float(epe_mean),
        safe_float(abs_rel_mean),
        safe_float(delta1_mean),
        values["num_pixels"],
    ])

```

```

def save_time_metrics(time_info, out_path):
    with open(out_path, "w", newline="") as f:
        writer = csv.writer(f)

        writer.writerow(["metric", "value"])
        writer.writerow(["num_images", time_info["num_images"]])

```

```

        writer.writerow(["total_inference_time_sec",
safe_float(time_info["total_inference_time_sec"])]])
        writer.writerow(["mean_inference_time_per_image_sec",
safe_float(time_info["mean_inference_time_per_image_sec"])]])
        writer.writerow(["fps", safe_float(time_info["fps"])]])

```

```

# =====
# 10. EVALUACIÓN DEL MODELO
# =====

```

```

@torch.no_grad()
def evaluate_model(model, loader, sample_names, calib_paths,
device, config):
    model.eval()

    per_image_rows = []
    added_metric_rows = []

    skipped_nonfinite = 0
    skipped_invalid = 0

    total_inference_time = 0.0
    measured_images = 0

    for idx, (imgL, imgR, dispL) in enumerate(loader):
        sample_name = sample_names[idx]
        calib_path = calib_paths[idx]

        imgL = imgL.float().to(device, non_blocking=True)
        imgR = imgR.float().to(device, non_blocking=True)
        dispL = dispL.float().to(device, non_blocking=True)

        if (
            not torch.isfinite(imgL).all()
            or not torch.isfinite(imgR).all()
            or not torch.isfinite(dispL).all()
        ):
            skipped_nonfinite += 1

```

```

        print(f"[WARN] {sample_name}: entrada no finita. Se
omite.")
        continue

    torch.cuda.synchronize()
    start_time = time.perf_counter()

    pred = model(imgL, imgR)

    torch.cuda.synchronize()
    end_time = time.perf_counter()

    inference_time = end_time - start_time
    total_inference_time += inference_time
    measured_images += 1

    if isinstance(pred, (tuple, list)):
        pred = pred[-1]

    pred = pred.squeeze(1) if pred.ndim == 4 else pred

    if not torch.isfinite(pred).all():
        skipped_nonfinite += 1
        print(f"[WARN] {sample_name}: predicción no finita. Se
omite.")
        continue

    basic = compute_basic_metrics(
        pred_disp=pred,
        gt_disp=dispL,
        maxdisp=config["maxdisp"],
    )

    if not np.isfinite(basic["d1_all"]) or not
np.isfinite(basic["epe"]):
        skipped_invalid += 1
        print(f"[WARN] {sample_name}: métricas inválidas. Se
omite.")
        continue

```

```

per_image_rows.append({
    "sample": sample_name,
    "d1_all": basic["d1_all"],
    "epe": basic["epe"],
    "abs_rel": basic["abs_rel"],
    "delta1": basic["delta1"],
    "valid_pixels": basic["valid_pixels"],
})

added_rows = compute_added_metrics_rows(
    pred_disp=pred,
    gt_disp=dispL,
    sample_name=sample_name,
    calib_path=calib_path,
    config=config,
)

added_metric_rows.extend(added_rows)

print(
    f"{idx + 1:03d}/{len(sample_names)} | {sample_name} | "
    f"D1-all={basic['d1_all']:.6f} | "
    f"EPE={basic['epe']:.6f} | "
    f"AbsRel={basic['abs_rel']:.6f} | "
    f"delta1={basic['delta1']:.6f} | "
    f"t_inf={inference_time:.4f}s"
)

if len(per_image_rows) == 0:
    raise RuntimeError("La evaluación no produjo ninguna
    métrica válida.")

mean_time = (
    total_inference_time / measured_images
    if measured_images > 0
    else np.nan
)

fps = (
    measured_images / total_inference_time

```

```

        if total_inference_time > 0
        else np.nan
    )

time_info = {
    "num_images": measured_images,
    "total_inference_time_sec": total_inference_time,
    "mean_inference_time_per_image_sec": mean_time,
    "fps": fps,
}

print(f"Muestras omitidas por NaN/Inf: {skipped_nonfinite}")
print(f"Muestras omitidas por métricas inválidas:
{skipped_invalid}")
print(f"Tiempo total de inferencia: {total_inference_time:.4f}
s")
print(f"Tiempo medio por imagen: {mean_time:.4f} s")
print(f"FPS aproximados: {fps:.4f}")

return per_image_rows, added_metric_rows, time_info

# =====
# 11. EVALUACIÓN INDIVIDUAL
# =====

def get_output_dir_for_evaluation(experimento, tipo_pesos):
    exp_info = EXPERIMENTOS[experimento]

    if experimento == "preentrenado":
        return exp_info["results_dir"]

    if tipo_pesos == "mejor_d1":
        subfolder = "MejorD1Teorica"
    elif tipo_pesos == "ultima_epoca":
        subfolder = "UltimaEpoca"
    else:
        raise ValueError(f"Tipo de pesos no válido: {tipo_pesos}")

    return os.path.join(exp_info["results_dir"], subfolder)

```

```

def run_final_evaluation(
    experimento,
    tipo_pesos,
    config,
    stackhourglass,
    loader,
    sample_names,
    calib_paths,
):
    assert_gpu_available()

    if experimento not in EXPERIMENTOS:
        raise ValueError(f"Experimento inválido: {experimento}")

    exp_info = EXPERIMENTOS[experimento]

    if tipo_pesos not in exp_info["weights"]:
        raise ValueError(
            f"Tipo de pesos '{tipo_pesos}' no disponible para
experimento '{experimento}'"
        )

    ckpt_path = exp_info["weights"][tipo_pesos]

    if not os.path.exists(ckpt_path):
        raise FileNotFoundError(f"No existe el checkpoint:
{ckpt_path}")

    output_dir = get_output_dir_for_evaluation(experimento,
tipo_pesos)
    ensure_dir(output_dir)

    print("\n=====
==")
    print("EVALUACIÓN FINAL PSMNet")

```

```

print("=====
")
    print(f"Experimento: {experimento}")
    print(f"Tipo pesos: {tipo_pesos}")
    print(f"Checkpoint: {ckpt_path}")
    print(f"Salida:      {output_dir}")

print("=====
")

device = torch.device("cuda")

model = stackhourglass(config["maxdisp"]).to(device)

model = load_checkpoint_into_model(
    model=model,
    ckpt_path=ckpt_path,
    device=device,
)

per_image_rows, added_metric_rows, time_info = evaluate_model(
    model=model,
    loader=loader,
    sample_names=sample_names,
    calib_paths=calib_paths,
    device=device,
    config=config,
)

metrics_per_image_path = os.path.join(output_dir,
"metrics_per_image.csv")
metrics_summary_path = os.path.join(output_dir,
"metrics_summary.csv")
added_metrics_path = os.path.join(output_dir,
"metricas_añadidas.csv")
added_metrics_summary_path = os.path.join(output_dir,
"metricas_añadidas_resumen.csv")
time_metrics_path = os.path.join(output_dir,
"metricas_tiempo.csv")

```



```

save_metrics_per_image(per_image_rows, metrics_per_image_path)
save_metrics_summary(per_image_rows, metrics_summary_path)
save_added_metrics(added_metric_rows, added_metrics_path)
save_added_metrics_summary(added_metric_rows,
added_metrics_summary_path)
save_time_metrics(time_info, time_metrics_path)

evaluation_config = {
    "experimento": experimento,
    "tipo_pesos": tipo_pesos,
    "checkpoint": ckpt_path,
    "output_dir": output_dir,
    "kitti2015_split": "segunda mitad: índices 100-199,
imágenes _10 y _11",
    "calibracion": "fx y baseline calculados por imagen desde
P2/P3 o P_rect_02/P_rect_03. Sin fallback.",
    "maxdisp": config["maxdisp"],
    "max_depth_m": config["max_depth_m"],
    "distance_bins": {
        "cerca": f"0-{{config['max_depth_m'] / 3.0:.1f}} m",
        "media": f">{{config['max_depth_m'] / 3.0:.1f}}-{{2.0 *
config['max_depth_m'] / 3.0:.1f}} m",
        "lejos": f">{{2.0 * config['max_depth_m'] /
3.0:.1f}}-{{config['max_depth_m']:.1f}} m",
    },
    "time_metrics": "Tiempo medido solo en la llamada de
inferencia model(imgL, imgR), excluyendo carga de pesos, lectura de
datos y cálculo de métricas.",
    "repo_psmnet": config["psmnet_repo_drive"],
}

with open(os.path.join(output_dir, "evaluation_config.json"),
"w") as f:
    json.dump(evaluation_config, f, indent=2)

print("\n=====
==")
print("EVALUACIÓN FINALIZADA")

```

```

print("=====
")
    print(f"CSV por imagen guardado en: {metrics_per_image_path}")
    print(f"CSV resumen guardado en: {metrics_summary_path}")
    print(f"CSV métricas añadidas guardado en:
{added_metrics_path}")
    print(f"CSV resumen métricas añadidas guardado en:
{added_metrics_summary_path}")
    print(f"CSV métricas tiempo guardado en: {time_metrics_path}")
    print(f"Resultados guardados en: {output_dir}")

print("=====
")

    result = {
        "experimento": experimento,
        "tipo_pesos": tipo_pesos,
        "checkpoint": ckpt_path,
        "output_dir": output_dir,
        "metrics_per_image": metrics_per_image_path,
        "metrics_summary": metrics_summary_path,
        "metricas_añadidas": added_metrics_path,
        "metricas_añadidas_resumen": added_metrics_summary_path,
        "metricas_tiempo": time_metrics_path,
        "time_info": time_info,
    }

    del model
    torch.cuda.empty_cache()

    return result

# =====
# 12. EVALUAR TODO SEGUIDO
# =====

def run_all_final_evaluations(config):
    assert_gpu_available()

```

```

print("=====
")
    print("PREPARACIÓN GENERAL")

print("=====
")

kitti2015_root, calib_root = prepare_kitti2015(config)

stackhourglass, myImageFloder = import_psmnet_from_drive(
    config["psmnet_repo_drive"]
)

loader, sample_names, calib_paths = make_eval_loader(
    myImageFloder=myImageFloder,
    kitti2015_root=kitti2015_root,
    calib_root=calib_root,
    config=config,
)

evaluaciones = [
    ("preentrenado", "preentrenado"),

    ("todas_capas", "mejor_d1"),
    ("todas_capas", "ultima_epoca"),

    ("intermedias_finales", "mejor_d1"),
    ("intermedias_finales", "ultima_epoca"),

    ("solo_finales", "mejor_d1"),
    ("solo_finales", "ultima_epoca"),
]

resultados = []

for experimento, tipo_pesos in evaluaciones:

```

```

print("\n\n#####
###")
    print(f"INICIANDO EVALUACIÓN: {experimento} |
{tipo_pesos}")

print("#####
\n")

    resultado = run_final_evaluation(
        experimento=experimento,
        tipo_pesos=tipo_pesos,
        config=config,
        stackhourglass=stackhourglass,
        loader=loader,
        sample_names=sample_names,
        calib_paths=calib_paths,
    )

    resultados.append(resultado)

resumen_global_path = os.path.join(
    config["output_base_dir"],
    "resumen_evaluaciones_finales.json"
)

with open(resumen_global_path, "w") as f:
    json.dump(resultados, f, indent=2)

print("\n=====
==")
    print("TODAS LAS EVALUACIONES FINALIZADAS")

print("=====
")
    print(f"Resumen global guardado en: {resumen_global_path}")

print("=====
")

```

```
    return resultados

# =====
# 13. EJECUCIÓN
# =====

try:
    resultados = run_all_final_evaluations(CONFIG)
    print(resultados)
finally:
    runtime.unassign()
```



7.4. ANEXO 4: COMPARATIVA MAPAS DE DISPARIDAD

```
# =====
# COMPARACIÓN VISUAL:
#   - Imagen RGB real KITTI2015
#   - Disparidad real KITTI
#   - Predicción método clásico
#   - Error método clásico
#   - Predicción PSMNet
#   - Error PSMNet
#
# Salidas:
#
#   /content/drive/MyDrive/TFG/Evaluacion/ComparacionMetodoclasicoVs
#   Cnn/FigurasComparacion/Horizontal/
#
#   /content/drive/MyDrive/TFG/Evaluacion/ComparacionMetodoclasicoVs
#   Cnn/FigurasComparacion/Vertical/
# =====

import os
import sys
import json
import shutil
import zipfile
import copy
from pathlib import Path

import cv2
import numpy as np
import torch
import matplotlib.pyplot as plt

from PIL import Image
from google.colab import drive
```

```

# =====
# 1. CONFIGURACIÓN
# =====

drive.mount("/content/drive", force_remount=True)

CONFIG = {
    "kitti2015_zip_drive":
        "/content/drive/MyDrive/TFG/Datasets/KITTI2015/data_scene_flow.zip",

    "psmnet_repo_drive":
        "/content/drive/MyDrive/TFG/PSMNet-master",

    "psmnet_weights_path":
        "/content/drive/MyDrive/TFG/Evaluacion/ComparacionMetodoClasicoVsCnn/pesos_seleccionados.tar",

    "classic_pred_npy_dir":
        "/content/drive/MyDrive/TFG/Evaluacion/MetodoClasico/Resultados/MapasDisparidad/PredNpy",

    "output_dir":
        "/content/drive/MyDrive/TFG/Evaluacion/ComparacionMetodoClasicoVsCnn/FigurasComparacion",

    "maxdisp": 192,
    "error_visual_max_px": 20.0,

    "figure_dpi": 200,

    # "auto" -> GPU si existe, si no CPU
    # "cuda" -> exige GPU
    # "cpu" -> fuerza CPU
    "device": "auto",

    "auto_disconnect": False,
}

```

```

SAMPLES = [
    "000163_10.png",
    "000168_10.png",
    "000169_10.png",
    "000171_10.png",
    "000178_10.png",
    "000186_10.png",
    "000198_10.png",
]

# =====
# 2. UTILIDADES GENERALES
# =====

def ensure_dir(path):
    Path(path).mkdir(parents=True, exist_ok=True)

def remove_dir_if_exists(path):
    p = Path(path)
    if p.exists():
        shutil.rmtree(p)

def get_device(config):
    requested = config.get("device", "auto").lower()

    if requested == "cuda":
        if not torch.cuda.is_available():
            raise RuntimeError(
                "Se ha solicitado device='cuda', pero no hay GPU
activa. "
                "Activa GPU en Colab o cambia CONFIG['device'] a
'auto' o 'cpu'."
            )
        device = torch.device("cuda")

    elif requested == "cpu":
        device = torch.device("cpu")

```



```

elif requested == "auto":
    if torch.cuda.is_available():
        device = torch.device("cuda")
    else:
        device = torch.device("cpu")

else:
    raise ValueError("CONFIG['device'] debe ser 'auto', 'cuda'
o 'cpu'.")

print("=====
====")
print("DISPOSITIVO DE EJECUCIÓN")

print("=====
====")
print(f"Device seleccionado: {device}")

if device.type == "cuda":
    print(f"GPU: {torch.cuda.get_device_name(0)}")
else:
    print("[AVISO] No se está usando GPU. PSMNet se ejecutará
en CPU y puede tardar bastante más.")

print("=====
====")

return device

def copy_zip_from_drive_to_content(src_drive_path,
dst_dir="/content/datasets_zip"):
    ensure_dir(dst_dir)

    src = Path(src_drive_path)

    if not src.exists():

```

```

        raise FileNotFoundError(f"No existe el ZIP en Drive:
{src_drive_path}")

dst = Path(dst_dir) / src.name

print("Copiando ZIP desde Drive:")
print(f" Origen: {src}")
print(f" Destino: {dst}")

shutil.copy2(src, dst)

if not dst.exists() or dst.stat().st_size == 0:
    raise RuntimeError(f"Error al copiar ZIP: {dst}")

return str(dst)

def unzip_file(zip_path, extract_to, clean_before_extract=True):
    if clean_before_extract:
        remove_dir_if_exists(extract_to)

    ensure_dir(extract_to)

    print("Descomprimiendo:")
    print(f" ZIP: {zip_path}")
    print(f" Destino: {extract_to}")

    try:
        with zipfile.ZipFile(zip_path, "r") as zf:
            bad_member = zf.testzip()

            if bad_member is not None:
                raise RuntimeError(
                    f"ZIP corrupto o incompleto. Primer archivo
problemático: {bad_member}"
                )

            zf.extractall(extract_to)

    except zipfile.BadZipFile:

```

```

        raise RuntimeError(f"Archivo ZIP no válido: {zip_path}")

    return extract_to

def find_kitti2015_training_root(search_root):
    required = {"image_2", "image_3", "disp_occ_0"}

    for root, dirs, files in os.walk(search_root):
        if required.issubset(set(dirs)):
            return root + "/"

    raise FileNotFoundError(
        f"No se encontró la raíz training de KITTI2015 dentro de {search_root}"
    )

def prepare_kitti2015(config):
    local_zip_dir = "/content/datasets_zip"

    kitti_zip = copy_zip_from_drive_to_content(
        config["kitti2015_zip_drive"],
        local_zip_dir
    )

    kitti_extract = unzip_file(
        kitti_zip,
        "/content/datasets/kitti2015",
        clean_before_extract=True
    )

    kitti_root = find_kitti2015_training_root(kitti_extract)

    print("KITTI2015 preparado correctamente.")
    print(f"Raíz imágenes/GT: {kitti_root}")

    return kitti_root

```

```

# =====
# 3. IMPORTAR PSMNet DESDE DRIVE
# =====

def import_psmnet_from_drive(repo_drive_path):
    repo_dir = Path(repo_drive_path)

    if not repo_dir.exists():
        raise FileNotFoundError(f"No existe el repositorio PSMNet
en Drive: {repo_drive_path}")

    expected_files = [
        repo_dir / "models" / "__init__.py",
        repo_dir / "dataloader" / "preprocess.py",
    ]

    for f in expected_files:
        if not f.exists():
            raise FileNotFoundError(f"Falta archivo esperado del
repo PSMNet: {f}")

    repo_dir_str = str(repo_dir.resolve())
    dataloader_dir_str = str((repo_dir / "dataloader").resolve())
    models_dir_str = str((repo_dir / "models").resolve())

    for p in [repo_dir_str, dataloader_dir_str, models_dir_str]:
        if p not in sys.path:
            sys.path.insert(0, p)

    from models import stackhourglass
    from dataloader import preprocess

    print(f"PSMNet importado correctamente desde Drive:
{repo_drive_path}")

    return stackhourglass, preprocess

```

```

# =====
# 4. CARGA DE PESOS PSMNet
# =====

def load_checkpoint_into_model(model, ckpt_path, device):
    if not os.path.exists(ckpt_path):
        raise FileNotFoundError(f"No existe el checkpoint: {ckpt_path}")

    if os.path.getsize(ckpt_path) == 0:
        raise RuntimeError(f"El checkpoint está vacío: {ckpt_path}")

    ckpt = torch.load(ckpt_path, map_location=device)

    if isinstance(ckpt, dict) and "state_dict" in ckpt:
        state = ckpt["state_dict"]
    else:
        state = ckpt

    if not isinstance(state, dict):
        raise RuntimeError(f"El checkpoint no contiene un state_dict válido: {ckpt_path}")

    clean_state = {}

    for k, v in state.items():
        clean_state[k.replace("module.", "")] = v

    model_state = model.state_dict()

    missing_keys = sorted(set(model_state.keys()) - set(clean_state.keys()))
    unexpected_keys = sorted(set(clean_state.keys()) - set(model_state.keys()))

    shape_mismatch = []

    for k in set(model_state.keys()) & set(clean_state.keys()):
        if model_state[k].shape != clean_state[k].shape:

```

```

        shape_mismatch.append(
            (k, tuple(clean_state[k].shape),
tuple(model_state[k].shape))
        )

    if missing_keys:
        msg = "\n".join(missing_keys[:30])
        raise RuntimeError(f"Faltan claves del checkpoint.
Primeras:\n{msg}")

    if unexpected_keys:
        msg = "\n".join(unexpected_keys[:30])
        raise RuntimeError(f"Hay claves inesperadas en el
checkpoint. Primeras:\n{msg}")

    if shape_mismatch:
        preview = "\n".join(
            [f"{k}: ckpt{a} vs model{b}" for k, a, b in
shape_mismatch[:30]]
        )
        raise RuntimeError(f"Hay tensores incompatibles en el
checkpoint. Primeros:\n{preview}")

    model.load_state_dict(clean_state, strict=True)

    print("Checkpoint PSMNet cargado correctamente:")
    print(f" {ckpt_path}")

    return model

def build_psmnet_model(config, stackhourglass, device):
    model = stackhourglass(config["maxdisp"]).to(device)

    model = load_checkpoint_into_model(
        model=model,
        ckpt_path=config["psmnet_weights_path"],
        device=device,
    )

```

```

    model.eval()

    return model

# =====
# 5. LECTURA DE IMÁGENES Y DISPARIDAD REAL
# =====

def get_sample_paths(kitti_root, sample_name):
    left_path = Path(kitti_root) / "image_2" / sample_name
    right_path = Path(kitti_root) / "image_3" / sample_name
    gt_path = Path(kitti_root) / "disp_occ_0" / sample_name

    if not left_path.exists():
        raise FileNotFoundError(f"No existe imagen izquierda:
{left_path}")

    if not right_path.exists():
        raise FileNotFoundError(f"No existe imagen derecha:
{right_path}")

    if not gt_path.exists():
        raise FileNotFoundError(f"No existe disparidad GT:
{gt_path}")

    return str(left_path), str(right_path), str(gt_path)

def read_rgb_image(path):
    img_bgr = cv2.imread(path, cv2.IMREAD_COLOR)

    if img_bgr is None:
        raise FileNotFoundError(f"No se pudo leer imagen RGB:
{path}")

    img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)

    return img_rgb

```

```

def read_kitti_disparity_png(path):
    disp_raw = cv2.imread(path, cv2.IMREAD_UNCHANGED)

    if disp_raw is None:
        raise FileNotFoundError(f"No se pudo leer disparidad GT: {path}")

    disp = disp_raw.astype(np.float32) / 256.0

    return disp

def read_classic_prediction_npy(classic_pred_npy_dir, sample_name):
    sample_stem = Path(sample_name).stem

    npy_path = Path(classic_pred_npy_dir) /
f"{sample_stem}_pred_disp_px.npy"

    if not npy_path.exists():
        raise FileNotFoundError(
            "No se encontró la predicción clásica .npy esperada:\n"
            f" {npy_path}\n"
            "Comprueba que antes has ejecutado el código del método"
            "clásico con guardado PredNpy."
        )

    pred = np.load(npy_path).astype(np.float32)

    return pred, str(npy_path)

# =====
# 6. INFERENCIA PSMNet PARA UNA IMAGEN
# =====

def pad_to_multiple_of_16(img_chw):
    _, h, w = img_chw.shape

    h_pad = int(np.ceil(h / 16.0) * 16)

```



```

w_pad = int(np.ceil(w / 16.0) * 16)

top_pad = h_pad - h
right_pad = w_pad - w

img_padded = np.pad(
    img_chw,
    ((0, 0), (top_pad, 0), (0, right_pad)),
    mode="constant",
    constant_values=0,
)

return img_padded, top_pad, right_pad

@torch.no_grad()
def predict_psmnet_disparity(model, preprocess_module, left_path,
    right_path, device):
    processed = preprocess_module.get_transform(augment=False)

    imgL = Image.open(left_path).convert("RGB")
    imgR = Image.open(right_path).convert("RGB")

    imgL = processed(imgL).numpy()
    imgR = processed(imgR).numpy()

    imgL, top_pad, right_pad = pad_to_multiple_of_16(imgL)
    imgR, top_pad_r, right_pad_r = pad_to_multiple_of_16(imgR)

    if top_pad != top_pad_r or right_pad != right_pad_r:
        raise RuntimeError("El padding de la imagen izquierda y
derecha no coincide.")

    imgL_t = torch.from_numpy(imgL).float().unsqueeze(0).to(device)
    imgR_t = torch.from_numpy(imgR).float().unsqueeze(0).to(device)

    pred = model(imgL_t, imgR_t)

    if isinstance(pred, (tuple, list)):
        pred = pred[-1]

```

```

if pred.ndim == 4:
    pred = pred.squeeze(1)

if pred.ndim == 3:
    pred = pred[0]

pred_np = pred.detach().cpu().numpy().astype(np.float32)

if top_pad > 0:
    pred_np = pred_np[top_pad:, :]

if right_pad > 0:
    pred_np = pred_np[:, :-right_pad]

return pred_np

# =====
# 7. VISUALIZACIÓN
# =====

def get_cmap_safe(name, fallback="viridis", bad_color="black"):
    try:
        cmap = plt.get_cmap(name)
    except Exception:
        cmap = plt.get_cmap(fallback)

    cmap = copy.copy(cmap)
    cmap.set_bad(color=bad_color)

    return cmap

def make_masked_disparity(disparity, maxdisp):
    valid = np.isfinite(disparity) & (disparity > 0.0) & (disparity <
float(maxdisp))
    disparity_masked = np.ma.array(disparity, mask=~valid)
    return disparity_masked

```

```

def make_masked_error(pred_disp, gt_disp, maxdisp):
    valid_gt = np.isfinite(gt_disp) & (gt_disp > 0.0) & (gt_disp <
float(maxdisp))

    err = np.abs(pred_disp - gt_disp).astype(np.float32)
    err = np.ma.array(err, mask=~valid_gt)

    return err, valid_gt

def compute_mean_error_px(pred_disp, gt_disp, maxdisp):
    valid_gt = np.isfinite(gt_disp) & (gt_disp > 0.0) & (gt_disp <
float(maxdisp))

    if valid_gt.sum() == 0:
        return np.nan

    err = np.abs(pred_disp[valid_gt] - gt_disp[valid_gt])

    if err.size == 0:
        return np.nan

    return float(np.mean(err))

def add_manual_colorbar(fig, im, x, y, panel_w, label,
tick_fontsize=9, label_fontsize=10):
    cbar_w = panel_w * 0.78
    cbar_h = 0.022
    cbar_x = x + (panel_w - cbar_w) / 2.0
    cbar_y = y - 0.055

    cax = fig.add_axes([cbar_x, cbar_y, cbar_w, cbar_h])

    cbar = fig.colorbar(
        im,
        cax=cax,
        orientation="horizontal",
    )

```

```

cbar.set_label(label, fontsize=label_fontsize)
cbar.ax.tick_params(labelsize=tick_fontsize)

return cbar

def add_image_panel(fig, x, y, panel_w, panel_h, title,
                    title_fontsize=14):
    ax = fig.add_axes([x, y, panel_w, panel_h])
    ax.set_title(title, fontsize=title_fontsize, fontweight="bold",
pad=6)
    ax.axis("off")

    return ax

def prepare_visual_arrays(gt_disp, classic_pred, psmnet_pred,
maxdisp):
    gt_disp_m = make_masked_disparity(gt_disp, maxdisp)
    classic_pred_m = make_masked_disparity(classic_pred, maxdisp)
    psmnet_pred_m = make_masked_disparity(psmnet_pred, maxdisp)

    classic_err_m, _ = make_masked_error(classic_pred, gt_disp,
maxdisp)
    psmnet_err_m, _ = make_masked_error(psmnet_pred, gt_disp,
maxdisp)

    classic_epe = compute_mean_error_px(classic_pred, gt_disp,
maxdisp)
    psmnet_epe = compute_mean_error_px(psmnet_pred, gt_disp,
maxdisp)

    return {
        "gt_disp_m": gt_disp_m,
        "classic_pred_m": classic_pred_m,
        "psmnet_pred_m": psmnet_pred_m,
        "classic_err_m": classic_err_m,
        "psmnet_err_m": psmnet_err_m,
        "classic_epe": classic_epe,

```

```

        "psmnet_epe": psmnet_epe,
    }

def validate_shapes(sample_name, gt_disp, classic_pred,
                    psmnet_pred):
    if gt_disp.shape != classic_pred.shape:
        raise RuntimeError(
            f"Forma incompatible GT vs clásico en {sample_name}: "
            f"GT={gt_disp.shape}, clásico={classic_pred.shape}"
        )

    if gt_disp.shape != psmnet_pred.shape:
        raise RuntimeError(
            f"Forma incompatible GT vs PSMNet en {sample_name}: "
            f"GT={gt_disp.shape}, PSMNet={psmnet_pred.shape}"
        )

# =====
# 8. GUARDADO FORMATO HORIZONTAL
# =====

def save_horizontal_comparison_figure(
    sample_name,
    rgb_img,
    gt_disp,
    classic_pred,
    psmnet_pred,
    output_dir,
    config,
):
    maxdisp = config["maxdisp"]
    error_vmax = config["error_visual_max_px"]

    validate_shapes(sample_name, gt_disp, classic_pred,
                    psmnet_pred)

    disp_cmap = get_cmap_safe("turbo", fallback="viridis",
                             bad_color="black")

```

```

err_cmap = get_cmap_safe("magma", fallback="inferno",
bad_color="black")

visual = prepare_visual_arrays(
    gt_disp=gt_disp,
    classic_pred=classic_pred,
    psmnet_pred=psmnet_pred,
    maxdisp=maxdisp,
)

fig_w = 24.0
fig_h = 9.0

fig = plt.figure(figsize=(fig_w, fig_h),
dpi=config["figure_dpi"])

left_margin = 0.025
right_margin = 0.025
horizontal_gap = 0.020

panel_w = (
    1.0
    - left_margin
    - right_margin
    - 3.0 * horizontal_gap
) / 4.0

x0 = left_margin
x1 = x0 + panel_w + horizontal_gap
x2 = x1 + panel_w + horizontal_gap
x3 = x2 + panel_w + horizontal_gap

img_h, img_w = rgb_img.shape[:2]
image_aspect = img_w / img_h

panel_h = (fig_w * panel_w) / (fig_h * image_aspect)

y_top = 0.690
y_mid = 0.415
y_bot = 0.160

```

```

ax_rgb = add_image_panel(fig, x0, y_mid, panel_w, panel_h,
"Imagen real KITTI2015")
ax_rgb.imshow(rgb_img, interpolation="nearest", aspect="auto")

ax_gt = add_image_panel(fig, x1, y_mid, panel_w, panel_h,
"Disparidad real KITTI")
im_gt = ax_gt.imshow(
    visual["gt_disp_m"],
    cmap=disp_cmap,
    vmin=0.0,
    vmax=float(maxdisp),
    interpolation="nearest",
    aspect="auto",
)
add_manual_colorbar(fig, im_gt, x1, y_mid, panel_w, "Disparidad
real (px)")

ax_classic_pred = add_image_panel(fig, x2, y_top, panel_w,
panel_h, "Método clásico: predicción")
im_classic_pred = ax_classic_pred.imshow(
    visual["classic_pred_m"],
    cmap=disp_cmap,
    vmin=0.0,
    vmax=float(maxdisp),
    interpolation="nearest",
    aspect="auto",
)
add_manual_colorbar(fig, im_classic_pred, x2, y_top, panel_w,
"Disparidad predicha (px)")

ax_classic_err = add_image_panel(
    fig,
    x3,
    y_top,
    panel_w,
    panel_h,
    f"Método clásico: error
|pred-GT|\nEPE={visual['classic_epe']:.3f} px",
)

```

```

im_classic_err = ax_classic_err.imshow(
    visual["classic_err_m"],
    cmap=err_cmap,
    vmin=0.0,
    vmax=float(error_vmax),
    interpolation="nearest",
    aspect="auto",
)
add_manual_colorbar(
    fig,
    im_classic_err,
    x3,
    y_top,
    panel_w,
    f"Error absoluto (px), saturado en {error_vmax:g}",
)

ax_psmnet_pred = add_image_panel(fig, x2, y_bot, panel_w,
panel_h, "PSMNet: predicción")
im_psmnet_pred = ax_psmnet_pred.imshow(
    visual["psmnet_pred_m"],
    cmap=disp_cmap,
    vmin=0.0,
    vmax=float(maxdisp),
    interpolation="nearest",
    aspect="auto",
)
add_manual_colorbar(fig, im_psmnet_pred, x2, y_bot, panel_w,
"Disparidad predicha (px)")

ax_psmnet_err = add_image_panel(
    fig,
    x3,
    y_bot,
    panel_w,
    panel_h,
    f"PSMNet: error |pred-GT|\nEPE={visual['psmnet_epe']:.3f}
px",
)
im_psmnet_err = ax_psmnet_err.imshow(

```



```

        visual["psmnet_err_m"],
        cmap=err_cmap,
        vmin=0.0,
        vmax=float(error_vmax),
        interpolation="nearest",
        aspect="auto",
    )
    add_manual_colorbar(
        fig,
        im_psmnet_err,
        x3,
        y_bot,
        panel_w,
        f"Error absoluto (px), saturado en {error_vmax:g}",
    )

    output_path = Path(output_dir) /
    f"{Path(sample_name).stem}_comparacion_horizontal.png"

    fig.savefig(output_path, bbox_inches="tight", pad_inches=0.15)
    plt.close(fig)

    return str(output_path)

# =====
# 9. GUARDADO FORMATO VERTICAL
# =====

def save_vertical_comparison_figure(
    sample_name,
    rgb_img,
    gt_disp,
    classic_pred,
    psmnet_pred,
    output_dir,
    config,
):
    maxdisp = config["maxdisp"]
    error_vmax = config["error_visual_max_px"]

```

```

    validate_shapes(sample_name, gt_disp, classic_pred,
psmnet_pred)

    disp_cmap = get_cmap_safe("turbo", fallback="viridis",
bad_color="black")
    err_cmap = get_cmap_safe("magma", fallback="inferno",
bad_color="black")

    visual = prepare_visual_arrays(
        gt_disp=gt_disp,
        classic_pred=classic_pred,
        psmnet_pred=psmnet_pred,
        maxdisp=maxdisp,
    )

    fig_w = 13.5
    fig_h = 16.0

    fig = plt.figure(figsize=(fig_w, fig_h),
dpi=config["figure_dpi"])

    horizontal_gap = 0.020

    left_margin = 0.070
    right_margin = 0.070

    panel_w = (
        1.0
        - left_margin
        - right_margin
        - horizontal_gap
    ) / 2.0

    x_left = left_margin
    x_right = x_left + panel_w + horizontal_gap
    x_center = (1.0 - panel_w) / 2.0

    img_h, img_w = rgb_img.shape[:2]
    image_aspect = img_w / img_h

```

```

panel_h = (fig_w * panel_w) / (fig_h * image_aspect)

y_img = 0.715
y_gt = 0.555
y_pred = 0.315
y_err = 0.075

# -----
# Imagen real arriba centrada
# -----

ax_rgb = add_image_panel(
    fig=fig,
    x=x_center,
    y=y_img,
    panel_w=panel_w,
    panel_h=panel_h,
    title="Imagen real KITTI2015",
)
ax_rgb.imshow(rgb_img, interpolation="nearest", aspect="auto")

# -----
# Disparidad real debajo centrada
# -----

ax_gt = add_image_panel(
    fig=fig,
    x=x_center,
    y=y_gt,
    panel_w=panel_w,
    panel_h=panel_h,
    title="Disparidad real KITTI",
)
im_gt = ax_gt.imshow(
    visual["gt_disp_m"],
    cmap=disp_cmap,
    vmin=0.0,
    vmax=float(maxdisp),
    interpolation="nearest",

```

```

        aspect="auto",
    )
add_manual_colorbar(
    fig=fig,
    im=im_gt,
    x=x_center,
    y=y_gt,
    panel_w=panel_w,
    label="Disparidad real (px)",
)

# -----
# Predicciones
# -----

ax_classic_pred = add_image_panel(
    fig=fig,
    x=x_left,
    y=y_pred,
    panel_w=panel_w,
    panel_h=panel_h,
    title="Método clásico: predicción",
)
im_classic_pred = ax_classic_pred.imshow(
    visual["classic_pred_m"],
    cmap=disp_cmap,
    vmin=0.0,
    vmax=float(maxdisp),
    interpolation="nearest",
    aspect="auto",
)
add_manual_colorbar(
    fig=fig,
    im=im_classic_pred,
    x=x_left,
    y=y_pred,
    panel_w=panel_w,
    label="Disparidad predicha (px)",
)

```

```

ax_psmnet_pred = add_image_panel(
    fig=fig,
    x=x_right,
    y=y_pred,
    panel_w=panel_w,
    panel_h=panel_h,
    title="PSMNet: predicción",
)
im_psmnet_pred = ax_psmnet_pred.imshow(
    visual["psmnet_pred_m"],
    cmap=disp_cmap,
    vmin=0.0,
    vmax=float(maxdisp),
    interpolation="nearest",
    aspect="auto",
)
add_manual_colorbar(
    fig=fig,
    im=im_psmnet_pred,
    x=x_right,
    y=y_pred,
    panel_w=panel_w,
    label="Disparidad predicha (px)",
)

# -----
# Errores
# -----

ax_classic_err = add_image_panel(
    fig=fig,
    x=x_left,
    y=y_err,
    panel_w=panel_w,
    panel_h=panel_h,
    title=f"Método clásico: error
|pred-GT|\nEPE={visual['classic_epe']:.3f} px",
)
im_classic_err = ax_classic_err.imshow(
    visual["classic_err_m"],

```

```

        cmap=err_cmap,
        vmin=0.0,
        vmax=float(error_vmax),
        interpolation="nearest",
        aspect="auto",
    )
    add_manual_colorbar(
        fig=fig,
        im=im_classic_err,
        x=x_left,
        y=y_err,
        panel_w=panel_w,
        label=f"Error absoluto (px), saturado en {error_vmax:g}",
    )

    ax_psmnet_err = add_image_panel(
        fig=fig,
        x=x_right,
        y=y_err,
        panel_w=panel_w,
        panel_h=panel_h,
        title=f"PSMNet: error
|pred-GT|\nEPE={visual['psmnet_epe']:.3f} px",
    )
    im_psmnet_err = ax_psmnet_err.imshow(
        visual["psmnet_err_m"],
        cmap=err_cmap,
        vmin=0.0,
        vmax=float(error_vmax),
        interpolation="nearest",
        aspect="auto",
    )
    add_manual_colorbar(
        fig=fig,
        im=im_psmnet_err,
        x=x_right,
        y=y_err,
        panel_w=panel_w,
        label=f"Error absoluto (px), saturado en {error_vmax:g}",
    )

```

```

    output_path = Path(output_dir) /
    f"{Path(sample_name).stem}_comparacion_vertical.png"

    fig.savefig(output_path, bbox_inches="tight", pad_inches=0.15)
    plt.close(fig)

    return str(output_path)

# =====
# 10. PROCESO COMPLETO
# =====

def run_visual_comparison(config, samples):
    output_base_dir = Path(config["output_dir"])
    horizontal_dir = output_base_dir / "Horizontal"
    vertical_dir = output_base_dir / "Vertical"

    ensure_dir(horizontal_dir)
    ensure_dir(vertical_dir)

    device = get_device(config)

    print("=====
    ===")
    print("COMPARACIÓN VISUAL MÉTODO CLÁSICO VS PSMNet")

    print("=====
    ===")
    print(f"Salida base: {output_base_dir}")
    print(f"Horizontal: {horizontal_dir}")
    print(f"Vertical: {vertical_dir}")

    print("=====
    ===")

    kitti_root = prepare_kitti2015(config)

```

```

stackhourglass, preprocess_module = import_psmnet_from_drive(
    config["psmnet_repo_drive"]
)

model = build_psmnet_model(
    config=config,
    stackhourglass=stackhourglass,
    device=device,
)

saved_figures = {
    "horizontal": [],
    "vertical": [],
}

for idx, sample_name in enumerate(samples, start=1):

    print("\n-----")
    print(f"{idx}/{len(samples)} - Procesando {sample_name}")

    print("-----")

    left_path, right_path, gt_path = get_sample_paths(
        kitti_root=kitti_root,
        sample_name=sample_name,
    )

    rgb_img = read_rgb_image(left_path)
    gt_disp = read_kitti_disparity_png(gt_path)

    classic_pred, classic_npy_path =
    read_classic_prediction_npy(
        classic_pred_npy_dir=config["classic_pred_npy_dir"],
        sample_name=sample_name,
    )

    print(f"Predicción clásica cargada desde:
    {classic_npy_path}")

```



```

psmnet_pred = predict_psmnet_disparity(
    model=model,
    preprocess_module=preprocess_module,
    left_path=left_path,
    right_path=right_path,
    device=device,
)

horizontal_path = save_horizontal_comparison_figure(
    sample_name=sample_name,
    rgb_img=rgb_img,
    gt_disp=gt_disp,
    classic_pred=classic_pred,
    psmnet_pred=psmnet_pred,
    output_dir=horizontal_dir,
    config=config,
)

vertical_path = save_vertical_comparison_figure(
    sample_name=sample_name,
    rgb_img=rgb_img,
    gt_disp=gt_disp,
    classic_pred=classic_pred,
    psmnet_pred=psmnet_pred,
    output_dir=vertical_dir,
    config=config,
)

saved_figures["horizontal"].append(horizontal_path)
saved_figures["vertical"].append(vertical_path)

print(f"Figura horizontal guardada en: {horizontal_path}")
print(f"Figura vertical guardada en: {vertical_path}")

config_out_path = output_base_dir /
"comparacion_visual_config.json"

config_to_save = dict(config)
config_to_save["samples"] = samples

```

```

config_to_save["device_used"] = str(device)
config_to_save["output_horizontal_dir"] = str(horizontal_dir)
config_to_save["output_vertical_dir"] = str(vertical_dir)
config_to_save["descripcion"] = (
    "Figuras comparativas con imagen RGB, disparidad real
KITTI, "
    "predicción del método clásico, error del método clásico, "
    "predicción PSMNet y error PSMNet."
)
config_to_save["layout_horizontal"] = (
    "Imagen real y disparidad real centradas verticalmente a la
izquierda. "
    "Método clásico en la parte superior derecha y PSMNet en la
parte inferior derecha. "
    "Todos los paneles de imagen se dibujan con el mismo tamaño
físico. "
    "El espacio horizontal entre todas las columnas es uniforme
y reducido. "
    "No se añade título global superior."
)
config_to_save["layout_vertical"] = (
    "Imagen real arriba centrada, disparidad real debajo
centrada, "
    "predicciones debajo con método clásico a la izquierda y
PSMNet a la derecha, "
    "y errores debajo con método clásico a la izquierda y
PSMNet a la derecha. "
    "Las posiciones verticales se han ajustado para reducir el
hueco tras la imagen real "
    "y aumentar la separación entre predicciones y errores,
manteniendo el tamaño global "
    "de la figura y el tamaño físico de los paneles."
)
config_to_save["nota_visualizacion"] = (
    "Las barras de color de disparidad usan escala 0-maxdisp. "
    "Las barras de error usan escala 0-error_visual_max_px y
saturan errores mayores. "
    "Las métricas EPE indicadas en los títulos se calculan
sobre píxeles con GT válido."
)

```

```

with open(config_out_path, "w") as f:
    json.dump(config_to_save, f, indent=2)

print("\n=====
=====")
    print("COMPARACIÓN VISUAL FINALIZADA")

print("=====
=====")
    print(f"Figuras horizontales guardadas:
{len(saved_figures['horizontal'])}")
    print(f"Figuras verticales guardadas:
{len(saved_figures['vertical'])}")
    print(f"Carpeta horizontal: {horizontal_dir}")
    print(f"Carpeta vertical:    {vertical_dir}")
    print(f"Config guardada en: {config_out_path}")

print("=====
=====")

    return saved_figures

```

```

# =====
# 11. EJECUCIÓN
# =====

try:
    figuras_generadas = run_visual_comparison(CONFIG, SAMPLES)

```

```
figuras_generadas

finally:
    if CONFIG.get("auto_disconnect", False):
        from google.colab import runtime
        runtime.unassign()
```



7.5. ANEXO 5: RECONSTRUCCIÓN 3D

```
# =====
# RECONSTRUCCIÓN 3D CON PSMNet - KITTI2015 TEST
# =====
# Pesos:
#
/content/drive/MyDrive/TFG/Reconstrucción3d/Pesos/pesos_seleccionad
os.tar
#
# Resultados:
#   /content/drive/MyDrive/TFG/Reconstrucción3d/Resultados
#
# Este código:
#   1. Procesa solo las imágenes indicadas manualmente.
#   2. Genera una imagen resumen vertical con:
#       imagen izquierda + disparidad + profundidad +
reconstrucción 3D.
#   3. Las vistas estáticas se renderizan viendo desde Z = 0 en
adelante.
#   4. La nube usa 1 de cada 2 puntos válidos.
#   5. El HTML no se muestra automáticamente en Colab.
# =====

# =====
# 0. INSTALACIÓN DE LIBRERÍAS
# =====

!pip -q install open3d plotly

# =====
# 1. IMPORTS
# =====

import os
import sys
import json
import time
```

```

import shutil
import zipfile
import subprocess
from pathlib import Path

import numpy as np
import cv2
from PIL import Image

import torch
import torch.nn.functional as F

import open3d as o3d
import plotly.graph_objects as go

import matplotlib.pyplot as plt

from google.colab import drive
from google.colab import runtime

# =====
# 2. MONTAJE DE DRIVE
# =====

drive.mount("/content/drive", force_remount=True)

# =====
# 3. CONFIGURACIÓN GENERAL
# =====

CONFIG = {
    # -----
    # Carpetas principales
    # -----
    "reconstruction_base_dir":
"/content/drive/MyDrive/TFG/Reconstrucción3d",
    "weights_dir":
"/content/drive/MyDrive/TFG/Reconstrucción3d/Pesos",

```

```

    "checkpoint_filename": "pesos_seleccionados.tar",
    "output_dir":
"/content/drive/MyDrive/TFG/Reconstrucción3d/Resultados",

# -----
# Dataset KITTI2015 test
# -----
    "kitti2015_zip_drive":
"/content/drive/MyDrive/TFG/Datasets/KITTI2015/data_scene_flow.zip"
,
    "kitti2015_calib_zip_drive":
"/content/drive/MyDrive/TFG/Datasets/KITTI2015/data_scene_flow_calib.zip",

# -----
# Repositorio PSMNet
# -----
    "psmnet_repo_drive":
"/content/drive/MyDrive/TFG/PSMNet-master",

# -----
# PSMNet
# -----
    "maxdisp": 192,

# -----
# Imágenes concretas de KITTI2015 test a procesar
# -----
    "selected_test_filenames": [
        "000017_10.png",
        "000084_10.png",
        "000105_10.png",
        "000174_10.png",
        "000179_10.png",
        "000199_10.png",
    ],
],

```

```

# -----
# Reconstrucción 3D
# -----
"min_disp": 1.0,
"min_depth_m": 1.0,
"max_depth_m": 80.0,

# 0.0 = no elimina nada.
# 0.15 = elimina el 15% superior para reducir cielo.
"remove_top_ratio_for_cloud": 0.0,

# -----
# Corrección visual de espejo
# -----
"flip_x_for_visual_match": True,

# -----
# Muestreo de puntos
# -----
# 2 significa: de cada 2 píxeles válidos, se usa 1.
"point_stride_ply": 2,
"point_stride_html": 2,

# No se hace recorte aleatorio adicional.
"max_points_html": None,
"max_points_static": None,

"random_seed": 42,

# -----
# HTML interactivo
# -----
"html_cube_size_m": 80.0,
"html_cube_center_x_m": 0.0,
"html_cube_center_y_m": 5.0,
"html_cube_center_z_m": 40.0,

"plotly_camera_eye_x": 0.0,
"plotly_camera_eye_y": 0.05,
"plotly_camera_eye_z": -2.8,

```



```

# El HTML NO se muestra automáticamente en Colab.
"display_first_html": False,

# -----
# Vista estática renderizada con cámara virtual
# -----
"render_width_px": 1600,
"render_height_px": 430,

# Mayor FOV = vista más abierta.
# Menor FOV = más zoom.
"render_fov_deg": 78.0,

# Cámara virtual. Se coloca detrás de Z=0 y mira hacia delante.
"render_camera_eye_x": 0.0,
"render_camera_eye_y": 1.45,
"render_camera_eye_z": -8.0,

# Mira hacia una zona lejana para que se vea desde Z=0 en
adelante.
"render_camera_target_x": 0.0,
"render_camera_target_y": 1.25,
"render_camera_target_z": 40.0,

# Desplazamiento vertical del encuadre.
"render_principal_y_offset_px": 20,

# Radio de cada punto en la imagen estática.
"render_point_radius": 1,

# Fondo.
"render_background_rgb": [245, 248, 252],

# Recorte de render. Ahora empieza en Z=0.
"render_min_depth_m": 0.0,
"render_max_depth_m": 80.0,
"render_max_abs_x_m": 45.0,
"render_min_y_m": -4.0,
"render_max_y_m": 20.0,

```

```

    # Coloreado rojo por cercanía.
    "red_depth_min_m": 0.0,
    "red_depth_max_m": 80.0,
}

# =====
# 4. UTILIDADES GENERALES
# =====

def ensure_dir(path):
    Path(path).mkdir(parents=True, exist_ok=True)

def remove_dir_if_exists(path):
    p = Path(path)
    if p.exists():
        shutil.rmtree(p)

def assert_gpu_available():
    if not torch.cuda.is_available():
        raise RuntimeError(
            "No hay GPU activa. En Colab ve a Entorno de ejecución
> "
            "Cambiar tipo de entorno de ejecución > GPU."
        )

def print_gpu_info():
    print("Comprobando GPU...")

    if torch.cuda.is_available():
        print(f"GPU activa: {torch.cuda.get_device_name(0)}")

    try:
        subprocess.run(["nvidia-smi"], check=False)
    except Exception as e:
        print(f"[WARN] No se pudo ejecutar nvidia-smi: {e}")

```

```

else:
    print("[WARN] No hay GPU activa.")

def copy_zip_from_drive_to_content(src_drive_path,
dst_dir="/content/datasets_zip"):
    ensure_dir(dst_dir)

    src = Path(src_drive_path)

    if not src.exists():
        raise FileNotFoundError(f"No existe el ZIP en Drive:
{src_drive_path}")

    dst = Path(dst_dir) / src.name

    print("Copiando ZIP desde Drive:")
    print(f" Origen: {src}")
    print(f" Destino: {dst}")

    shutil.copy2(src, dst)

    if not dst.exists() or dst.stat().st_size == 0:
        raise RuntimeError(f"Error al copiar el ZIP: {dst}")

    return str(dst)

def unzip_file(zip_path, extract_to, clean_before_extract=True):
    if clean_before_extract:
        remove_dir_if_exists(extract_to)

    ensure_dir(extract_to)

    print("Descomprimiendo ZIP:")
    print(f" ZIP: {zip_path}")
    print(f" Destino: {extract_to}")

    try:
        with zipfile.ZipFile(zip_path, "r") as zf:

```

```

        bad_member = zf.testzip()

        if bad_member is not None:
            raise RuntimeError(
                f"ZIP corrupto o incompleto. Primer archivo
problemático: {bad_member}"
            )

        zf.extractall(extract_to)

    except zipfile.BadZipFile:
        raise RuntimeError(f"Archivo ZIP no válido: {zip_path}")

    return extract_to


def safe_float_for_json(x):
    if x is None:
        return None

    try:
        x = float(x)
    except Exception:
        return str(x)

    if not np.isfinite(x):
        return None

    return x


def get_cv2_colormap(name, fallback):
    if hasattr(cv2, name):
        return getattr(cv2, name)

    return fallback

```

```

# =====
# 5. CHECKPOINT
# =====

def get_selected_checkpoint_path(config):
    weights_dir = Path(config["weights_dir"])
    checkpoint_filename = config["checkpoint_filename"]

    if not weights_dir.exists():
        raise FileNotFoundError(
            f"No existe la carpeta de pesos:\n{weights_dir}\n\n"
            "Crea esta carpeta y coloca dentro el archivo"
            pesos_seleccionados.tar."
        )

    ckpt_path = weights_dir / checkpoint_filename

    if not ckpt_path.exists():
        raise FileNotFoundError(
            f"No existe el checkpoint"
            f"seleccionado:\n{ckpt_path}\n\n"
            "El archivo debe llamarse exactamente:"
            pesos_seleccionados.tar"
        )

    if ckpt_path.stat().st_size == 0:
        raise RuntimeError(
            f"El checkpoint seleccionado está vacío:\n{ckpt_path}"
        )

    print("Checkpoint seleccionado:")
    print(f"    {ckpt_path}")

    return str(ckpt_path)

```

```

# =====
# 6. PREPARACIÓN DE KITTI2015 TEST
# =====

def find_kitti2015_testing_root(search_root):
    search_root = Path(search_root)

    preferred = search_root / "testing"

    if (preferred / "image_2").exists() and (preferred /
"image_3").exists():
        return str(preferred) + "/"

    for root, dirs, files in os.walk(search_root):
        dirs_set = set(dirs)

        if {"image_2", "image_3"}.issubset(dirs_set):
            root_path = Path(root)

            if root_path.name == "testing" or "testing" in
root_path.parts:
                return str(root_path) + "/"

            raise FileNotFoundError(
                f"No se encontró la raíz testing de KITTI2015 dentro de
{search_root}"
            )

def find_kitti2015_testing_calib_root(search_root):
    search_root = Path(search_root)

    preferred = search_root / "testing" / "calib_cam_to_cam"

    if preferred.exists():
        txt_files = list(preferred.glob("*.txt"))

        numeric_txt = [
            f for f in txt_files
            if len(f.name) >= 10 and f.name[:6].isdigit()

```

```

    ]

    if len(numeric_txt) > 0:
        return str(preferred) + "/"

candidates = []

for root, dirs, files in os.walk(search_root):
    root_path = Path(root)

    if "testing" not in root_path.parts:
        continue

    txt_files = [f for f in files if f.endswith(".txt")]

    numeric_txt = [
        f for f in txt_files
        if len(f) >= 10 and f[:6].isdigit()
    ]

    if len(numeric_txt) > 0:
        candidates.append(root)

if not candidates:
    raise FileNotFoundError(
        f"No se encontró carpeta de calibración de KITTI2015
testing dentro de {search_root}"
    )

candidates = sorted(candidates)

return candidates[0] + "/"

def prepare_kitti2015_test(config):
    local_zip_dir = "/content/datasets_zip"

    kitti_zip = copy_zip_from_drive_to_content(
        config["kitti2015_zip_drive"],
        local_zip_dir
    )

```

```

)

kitti_extract = unzip_file(
    kitti_zip,
    "/content/datasets/kitti2015",
    clean_before_extract=True
)

kitti_testing_root = find_kitti2015_testing_root(kitti_extract)

calib_zip = copy_zip_from_drive_to_content(
    config["kitti2015_calib_zip_drive"],
    local_zip_dir
)

calib_extract = unzip_file(
    calib_zip,
    "/content/datasets/kitti2015_calib",
    clean_before_extract=True
)

calib_testing_root =
find_kitti2015_testing_calib_root(calib_extract)

print("KITTI2015 test preparado correctamente.")
print(f"Raíz imágenes test:      {kitti_testing_root}")
print(f"Raíz calibración test:   {calib_testing_root}")

return kitti_testing_root, calib_testing_root

def build_kitti2015_test_pairs(kitti_testing_root,
calib_testing_root, config):
    left_dir = Path(kitti_testing_root) / "image_2"
    right_dir = Path(kitti_testing_root) / "image_3"
    calib_dir = Path(calib_testing_root)

    if not left_dir.exists():
        raise FileNotFoundError(f"No existe image_2: {left_dir}")

```



```

if not right_dir.exists():
    raise FileNotFoundError(f"No existe image_3: {right_dir}")

if not calib_dir.exists():
    raise FileNotFoundError(f"No existe carpeta de calibración:
{calib_dir}")

requested = config["selected_test_filenames"]

left_names = {p.name for p in left_dir.glob("*.png")}
right_names = {p.name for p in right_dir.glob("*.png")}
calib_names = {p.name for p in calib_dir.glob("*.txt")}

samples = []
missing = []

for name in requested:
    base_id = name[:6]
    calib_name = f"{base_id}.txt"

    ok = (
        name in left_names
        and name in right_names
        and calib_name in calib_names
    )

    if not ok:
        missing_reasons = []

        if name not in left_names:
            missing_reasons.append("falta en image_2")

        if name not in right_names:
            missing_reasons.append("falta en image_3")

        if calib_name not in calib_names:
            missing_reasons.append(f"falta calibración
{calib_name}")

        missing.append((name, ", ".join(missing_reasons)))

```

```

        continue

    samples.append({
        "name": name,
        "stem": Path(name).stem,
        "left_path": str(left_dir / name),
        "right_path": str(right_dir / name),
        "calib_path": str(calib_dir / calib_name),
    })

    if len(missing) > 0:
        print("[WARN] Algunas imágenes solicitadas no se han
encontrado completas:")

        for name, reason in missing:
            print(f" - {name}: {reason}")

    if len(samples) == 0:
        raise RuntimeError("No se encontró ninguna imagen
solicitada válida en KITTI2015 test.")

    print(f"Pares estéreo seleccionados para reconstrucción:
{len(samples)}")

    for s in samples:
        print(f" {s['name']}")

    return samples

# =====
# 7. IMPORTAR PSMNet
# =====

def import_psmnet(config):
    repo_drive = Path(config["psmnet_repo_drive"])

    if repo_drive.exists():
        repo_dir = repo_drive
        print(f"Usando repositorio PSMNet desde Drive: {repo_dir}")

```

```

else:
    print("[WARN] No se encontró PSMNet en Drive.")
    print("Se intentará clonar el repositorio oficial en
/content/PSMNet.")

    os.chdir("/content")
    !rm -rf /content/PSMNet
    !git clone https://github.com/JiaRenChang/PSMNet.git

    repo_dir = Path("/content/PSMNet")

    if not repo_dir.exists():
        raise RuntimeError("No se pudo clonar PSMNet.")

expected_files = [
    repo_dir / "models" / "__init__.py",
    repo_dir / "dataloader" / "preprocess.py",
]

for f in expected_files:
    if not f.exists():
        raise FileNotFoundError(f"Falta archivo esperado del
repo PSMNet: {f}")

repo_dir_str = str(repo_dir.resolve())
dataloader_dir_str = str((repo_dir / "dataloader").resolve())
models_dir_str = str((repo_dir / "models").resolve())

for p in [repo_dir_str, dataloader_dir_str, models_dir_str]:
    if p not in sys.path:
        sys.path.insert(0, p)

from models import stackhourglass
from dataloader import preprocess

print("PSMNet importado correctamente.")

return stackhourglass, preprocess, str(repo_dir)

```

```

# =====
# 8. CARGA DE CHECKPOINT
# =====

def load_checkpoint_into_model(model, ckpt_path, device):
    ckpt_path = str(ckpt_path)

    if not os.path.exists(ckpt_path):
        raise FileNotFoundError(f"No existe el checkpoint: {ckpt_path}")

    if os.path.getsize(ckpt_path) == 0:
        raise RuntimeError(f"El checkpoint está vacío: {ckpt_path}")

    print("Cargando checkpoint:")
    print(f"  {ckpt_path}")

    ckpt = torch.load(ckpt_path, map_location=device)

    if isinstance(ckpt, dict) and "state_dict" in ckpt:
        state = ckpt["state_dict"]
    else:
        state = ckpt

    if not isinstance(state, dict):
        raise RuntimeError(f"El checkpoint no contiene un state_dict válido: {ckpt_path}")

    clean_state = {}

    for k, v in state.items():
        clean_state[k.replace("module.", "")] = v

    model_state = model.state_dict()

    missing_keys = sorted(set(model_state.keys()) -
                           set(clean_state.keys()))
    unexpected_keys = sorted(set(clean_state.keys()) -
                             set(model_state.keys()))

```

```

shape_mismatch = []

for k in set(model_state.keys()) & set(clean_state.keys()):
    if model_state[k].shape != clean_state[k].shape:
        shape_mismatch.append(
            (k, tuple(clean_state[k].shape),
tuple(model_state[k].shape))

        )

    if missing_keys:
        msg = "\n".join(missing_keys[:30])
        raise RuntimeError(
            f"Faltan claves del checkpoint al cargar el modelo.
Primeras:\n{msg}"
        )

    if unexpected_keys:
        msg = "\n".join(unexpected_keys[:30])
        raise RuntimeError(
            f"Hay claves inesperadas en el checkpoint.
Primeras:\n{msg}"
        )

    if shape_mismatch:
        preview = "\n".join(
            [f"{k}: ckpt{a} vs model{b}" for k, a, b in
shape_mismatch[:30]]
        )
        raise RuntimeError(
            f"Hay tensores con forma incompatible en el checkpoint.
Primeros:\n{preview}"
        )

    model.load_state_dict(clean_state, strict=True)

    print("Checkpoint cargado correctamente con verificación
estricta.")

    return model

```

```

# =====
# 9. CALIBRACIÓN KITTI
# =====

def parse_kitti_calib_file(calib_path):
    data = {}

    with open(calib_path, "r") as f:
        for line in f:
            if ":" not in line:
                continue

            key, value = line.split(":", 1)
            key = key.strip()
            value = value.strip()

            try:
                values = [float(x) for x in value.split()]
            except ValueError:
                continue

            data[key] = values

    return data

def get_projection_matrix_from_candidates(data, candidates,
calib_path):
    for key in candidates:
        if key in data:
            values = data[key]

            if len(values) != 12:
                raise RuntimeError(
                    f"La clave {key} existe en {calib_path}, "
                    f"pero no tiene 12 valores. Tiene {len(values)}"
                    valores."
                )

            P = np.array(values, dtype=np.float64).reshape(3, 4)

```

```

        return key, P

available_keys = sorted(list(data.keys()))

raise KeyError(
    "No se encontró ninguna matriz de proyección válida.\n"
    f"Archivo: {calib_path}\n"
    f"Claves buscadas: {candidates}\n"
    f"Claves disponibles en el archivo: {available_keys}"
)

def get_camera_params_from_kitti_calib(calib_path):
    data = parse_kitti_calib_file(calib_path)

    left_candidates = [
        "P2",
        "P_rect_02",
        "P_rect_2",
    ]

    right_candidates = [
        "P3",
        "P_rect_03",
        "P_rect_3",
    ]

    left_key, P_left = get_projection_matrix_from_candidates(
        data=data,
        candidates=left_candidates,
        calib_path=calib_path
    )

    right_key, P_right = get_projection_matrix_from_candidates(
        data=data,
        candidates=right_candidates,
        calib_path=calib_path
    )

```

```

fx = P_left[0, 0]
fy = P_left[1, 1]
cx = P_left[0, 2]
cy = P_left[1, 2]

tx_left = -P_left[0, 3] / P_left[0, 0]
tx_right = -P_right[0, 3] / P_right[0, 0]

baseline = abs(tx_right - tx_left)

if fx <= 0 or fy <= 0 or baseline <= 0:
    raise RuntimeError(
        f"Calibración inválida en {calib_path}.\n"
        f"Clave izquierda usada: {left_key}\n"
        f"Clave derecha usada: {right_key}\n"
        f"fx={fx}, fy={fy}, baseline={baseline}"
    )

print(f"Claves de calibración usadas: izquierda={left_key},
derecha={right_key}")

return {
    "fx": float(fx),
    "fy": float(fy),
    "cx": float(cx),
    "cy": float(cy),
    "baseline": float(baseline),
    "left_projection_key": left_key,
    "right_projection_key": right_key,
    "P_left": P_left,
    "P_right": P_right,
}

```



```

# =====
# 10. INFERENCIA PSMNet
# =====

def pad_tensor_for_psmnet(img_tensor):
    _, _, h, w = img_tensor.shape

    if h <= 384 and w <= 1248:
        target_h = 384
        target_w = 1248
    else:
        target_h = int(np.ceil(h / 16.0) * 16)
        target_w = int(np.ceil(w / 16.0) * 16)

    top_pad = target_h - h
    right_pad = target_w - w

    if top_pad < 0 or right_pad < 0:
        raise RuntimeError(
            f"Padding inválido. Imagen {h}x{w}, target
{target_h}x{target_w}"
        )

    img_padded = F.pad(
        img_tensor,
        pad=(0, right_pad, top_pad, 0),
        mode="constant",
        value=0
    )

    pad_info = {
        "original_h": h,
        "original_w": w,
        "target_h": target_h,
        "target_w": target_w,
        "top_pad": top_pad,
        "right_pad": right_pad,
    }

    return img_padded, pad_info

```

```

@torch.no_grad()
def predict_disparity_psmnet(model, preprocess_module, left_path,
right_path, device):
    img_left_pil = Image.open(left_path).convert("RGB")
    img_right_pil = Image.open(right_path).convert("RGB")

    original_w, original_h = img_left_pil.size

    if img_right_pil.size != img_left_pil.size:
        raise RuntimeError(
            f"Las imágenes izquierda y derecha tienen distinto
tamaño: "
            f"{img_left_pil.size} vs {img_right_pil.size}"
        )

    transform = preprocess_module.get_transform(augment=False)

    img_left = transform(img_left_pil).unsqueeze(0).float()
    img_right = transform(img_right_pil).unsqueeze(0).float()

    img_left, pad_info = pad_tensor_for_psmnet(img_left)
    img_right, _ = pad_tensor_for_psmnet(img_right)

    img_left = img_left.to(device, non_blocking=True)
    img_right = img_right.to(device, non_blocking=True)

    if device.type == "cuda":
        torch.cuda.synchronize()

    start_time = time.perf_counter()

    pred = model(img_left, img_right)

    if device.type == "cuda":
        torch.cuda.synchronize()

    inference_time = time.perf_counter() - start_time

```

```

if isinstance(pred, (tuple, list)):
    pred = pred[-1]

pred = pred.squeeze()

if pred.ndim != 2:
    raise RuntimeError(f"La predicción de disparidad tiene
forma inesperada: {pred.shape}")

top_pad = pad_info["top_pad"]
h = pad_info["original_h"]
w = pad_info["original_w"]

pred = pred[top_pad:top_pad + h, :w]

pred_disp = pred.detach().cpu().numpy().astype(np.float32)

if pred_disp.shape != (original_h, original_w):
    raise RuntimeError(
        f"La disparidad final no coincide con el tamaño
original. "
        f"Disp: {pred_disp.shape}, original: {(original_h,
original_w)}"
    )

return pred_disp, inference_time, pad_info

# =====
# 11. MAPAS DE COLOR
# =====

def save_disparity_color_fixed_scale(disparity, out_path, config):
    valid = np.isfinite(disparity) & (disparity >= config["min_disp"])

    disparity_clip = np.clip(disparity, 0.0, float(config["maxdisp"]))
    disparity_uint8 = (disparity_clip / float(config["maxdisp"])) *
255.0).astype(np.uint8)

    colormap = get_cv2_colormap("COLORMAP_TURBO", cv2.COLORMAP_JET)

```

```

color_bgr = cv2.applyColorMap(disparity_uint8, colormap)
color_bgr[~valid] = (0, 0, 0)

cv2.imwrite(out_path, color_bgr)

def save_disparity_with_colorbar(disp, out_path, config,
title="Predicción de disparidad"):
    valid = np.isfinite(disp) & (disp >= config["min_disp"])

    disp_masked = np.ma.array(disp, mask=~valid)

    cmap = plt.get_cmap("turbo").copy()
    cmap.set_bad(color="black")

    fig, ax = plt.subplots(figsize=(12, 5))

    im = ax.imshow(
        disp_masked,
        cmap=cmap,
        vmin=0.0,
        vmax=float(config["maxdisp"])
    )

    ax.set_title(title, fontsize=22, fontweight="bold")
    ax.axis("off")

    cbar = plt.colorbar(
        im,
        ax=ax,
        orientation="horizontal",
        fraction=0.08,
        pad=0.16
    )

    cbar.set_label("Disparidad (píxeles)", fontsize=16)
    cbar.ax.tick_params(labelsize=13)

    ticks = list(range(0, int(config["maxdisp"]), 25))

```

```

cbar.set_ticks(ticks)

plt.tight_layout()
plt.savefig(out_path, dpi=200, bbox_inches="tight")
plt.close(fig)

def normalize_to_uint8(values, valid_mask, invert=False,
percentile_min=1, percentile_max=99):
    out = np.zeros(values.shape, dtype=np.uint8)

    if valid_mask.sum() == 0:
        return out

    valid_values = values[valid_mask]
    valid_values = valid_values[np.isfinite(valid_values)]

    if valid_values.size == 0:
        return out

    vmin = np.percentile(valid_values, percentile_min)
    vmax = np.percentile(valid_values, percentile_max)

    if not np.isfinite(vmin) or not np.isfinite(vmax) or abs(vmax -
vmin) < 1e-6:
        vmin = float(np.min(valid_values))
        vmax = float(np.max(valid_values))

    if abs(vmax - vmin) < 1e-6:
        return out

    norm = (values - vmin) / (vmax - vmin)
    norm = np.clip(norm, 0.0, 1.0)

    if invert:
        norm = 1.0 - norm

    out = (norm * 255.0).astype(np.uint8)
    out[~valid_mask] = 0

```

```

return out

def save_depth_color(depth, out_path, config, title="Estimación de
profundidad"):
    """
    Guarda una visualización de la profundidad con título y barra
    de color.

    La profundidad se muestra en metros. Se usa turbo_r para
    mantener la
    interpretación visual habitual de la versión anterior:
    - distancias pequeñas: colores más cálidos
    - distancias grandes: colores más fríos
    """

    valid = (
        np.isfinite(depth)
        & (depth >= config["min_depth_m"])
        & (depth <= config["max_depth_m"])
    )

    depth_masked = np.ma.array(depth, mask=~valid)

    cmap = plt.get_cmap("turbo_r").copy()
    cmap.set_bad(color="black")

    fig, ax = plt.subplots(figsize=(12, 5))

    im = ax.imshow(
        depth_masked,
        cmap=cmap,
        vmin=float(config["min_depth_m"]),
        vmax=float(config["max_depth_m"])
    )

    ax.set_title(title, fontsize=22, fontweight="bold")
    ax.axis("off")

    cbar = plt.colorbar(

```

```

        im,
        ax=ax,
        orientation="horizontal",
        fraction=0.08,
        pad=0.16
    )

    cbar.set_label("Profundidad (metros)", fontsize=16)
    cbar.ax.tick_params(labelsiz=13)

    max_depth = float(config["max_depth_m"])
    tick_step = 10.0 if max_depth <= 100.0 else 20.0
    ticks = np.arange(0.0, max_depth + tick_step, tick_step)
    ticks = ticks[(ticks >= float(config["min_depth_m"])) & (ticks
<= max_depth)]
    cbar.set_ticks(ticks)

    plt.tight_layout()
    plt.savefig(out_path, dpi=200, bbox_inches="tight")
    plt.close(fig)

# =====
# 12. DISPARIDAD A PROFUNDIDAD
# =====

def disparity_to_depth_m(disparity, fx, baseline, min_disp=1.0):
    disparity_safe = np.copy(disparity).astype(np.float32)

    invalid = ~np.isfinite(disparity_safe) | (disparity_safe < min_disp)

    disparity_safe[invalid] = np.nan

    depth = (fx * baseline) / disparity_safe

    depth[invalid] = np.nan

    return depth.astype(np.float32)

```

```

# =====
# 13. NUBE DE PUNTOS 3D
# =====

def make_point_cloud_from_disparity(
    disp,
    rgb_image,
    calib_params,
    config,
    point_stride=2
):
    fx = calib_params["fx"]
    fy = calib_params["fy"]
    cx = calib_params["cx"]
    cy = calib_params["cy"]
    baseline = calib_params["baseline"]

    h, w = disp.shape

    if rgb_image.shape[:2] != (h, w):
        raise RuntimeError(
            f"RGB y disparidad tienen tamaños distintos:
{rgb_image.shape[:2]} vs {(h, w)}"
        )

    depth = disparity_to_depth_m(
        disp=disp,
        fx=fx,
        baseline=baseline,
        min_disp=config["min_disp"]
    )

    u, v = np.meshgrid(np.arange(w), np.arange(h))

    Z = depth

    X_original = (u.astype(np.float32) - cx) * Z / fx

    if config["flip_x_for_visual_match"]:
        X = -X_original

```



```

else:
    X = X_original

Y = -(v.astype(np.float32) - cy) * Z / fy

valid = (
    np.isfinite(X)
    & np.isfinite(Y)
    & np.isfinite(Z)
    & np.isfinite(disparity)
    & (disparity >= config["min_disparity"])
    & (Z >= config["min_depth_m"])
    & (Z <= config["max_depth_m"])
)

if config["remove_top_ratio_for_cloud"] > 0:
    top_limit = int(h *
float(config["remove_top_ratio_for_cloud"]))
    valid[:top_limit, :] = False

if point_stride > 1:
    stride_mask = np.zeros_like(valid, dtype=bool)
    stride_mask[:, :point_stride, ::point_stride] = True
    valid = valid & stride_mask

points = np.stack([X[valid], Y[valid], Z[valid]],
axis=1).astype(np.float32)
colors = rgb_image[valid].astype(np.float32) / 255.0

return points, colors, depth, valid

def save_point_cloud_ply(points, colors, out_path):
    if points.shape[0] == 0:
        raise RuntimeError("No hay puntos válidos para guardar la
nube PLY.")

    pcd = o3d.geometry.PointCloud()

```

```

    pcd.points =
o3d.utility.Vector3dVector(points.astype(np.float64))
    pcd.colors =
o3d.utility.Vector3dVector(colors.astype(np.float64))

    ok = o3d.io.write_point_cloud(out_path, pcd)

    if not ok:
        raise RuntimeError(f"No se pudo guardar la nube de puntos:
{out_path}")

def sample_points_for_visualization(points, colors, max_points,
seed=42):
    n = points.shape[0]

    if n == 0:
        return points, colors

    if max_points is None or n <= max_points:
        return points, colors

    rng = np.random.default_rng(seed)
    idx = rng.choice(n, size=max_points, replace=False)

    return points[idx], colors[idx]

def colors_to_plotly_rgb_strings(colors):
    colors_uint8 = np.clip(colors * 255.0, 0, 255).astype(np.uint8)

    rgb_strings = [
        f"rgb({r},{g},{b}) "
        for r, g, b in colors_uint8
    ]

    return rgb_strings

def get_equal_cube_ranges(config, prefix):

```

```

cube_size = float(config[f"{prefix}_cube_size_m"])

cx = float(config[f"{prefix}_cube_center_x_m"])
cy = float(config[f"{prefix}_cube_center_y_m"])
cz = float(config[f"{prefix}_cube_center_z_m"])

half = cube_size / 2.0

x_range = [cx - half, cx + half]
y_range = [cy - half, cy + half]
z_range = [cz - half, cz + half]

return x_range, y_range, z_range

def save_interactive_point_cloud_html(points, colors, out_path,
config, title="Nube de puntos 3D"):
    """
    Guarda la nube de puntos interactiva en HTML.

    En esta versión:
    - Todos los puntos tienen el mismo tamaño.
    - Todos los puntos son opacos.
    - Se usa una única traza Scatter3d, como en la versión
    original.
    - La prioridad visual depende del renderizado 3D de
    Plotly/WebGL.

    Nota:
    Al ser una nube de puntos y no una malla cerrada, pueden
    seguir viéndose
    puntos del fondo a través de huecos entre puntos cercanos.
    Sin embargo,
    al usar opacity=1.0 se evita el efecto de transparencia
    artificial.
    """

    if points.shape[0] == 0:
        raise RuntimeError("No hay puntos válidos para crear HTML
    interactivo.")

```

```

finite = (
    np.isfinite(points).all(axis=1)
    & np.isfinite(colors).all(axis=1)
)

points = points[finite]
colors = colors[finite]

if points.shape[0] == 0:
    raise RuntimeError("No quedan puntos finitos para crear
HTML interactivo.")

rgb_strings = colors_to_plotly_rgb_strings(colors)

x_range, y_range, z_range = get_equal_cube_ranges(config,
prefix="html")

camera = dict(
    up=dict(x=0, y=1, z=0),
    center=dict(x=0, y=0, z=0),
    eye=dict(
        x=float(config["plotly_camera_eye_x"]),
        y=float(config["plotly_camera_eye_y"]),
        z=float(config["plotly_camera_eye_z"])
    )
)

fig = go.Figure(
    data=[
        go.Scatter3d(
            x=points[:, 0],
            y=points[:, 1],
            z=points[:, 2],
            mode="markers",
            showlegend=False,
            hoverinfo="skip",
            marker=dict(
                size=1.2,
                color=rgb_strings,

```

```

        opacity=1.0,
        line=dict(width=0)
    )
)
]
)

fig.update_layout(
    title=title,
    scene=dict(
        xaxis=dict(
            title="X lateral [m]",
            range=x_range,
            zeroline=True
        ),
        yaxis=dict(
            title="Y vertical [m]",
            range=y_range,
            zeroline=True
        ),
        zaxis=dict(
            title="Z profundidad [m]",
            range=z_range,
            zeroline=True
        ),
        aspectmode="cube",
        camera=camera
    ),
    margin=dict(l=0, r=0, t=40, b=0)
)

fig.write_html(out_path)

# =====
# 14. RENDER ESTÁTICO CON CÁMARA VIRTUAL
# =====

def normalize_vector(v):
    v = np.asarray(v, dtype=np.float64)
    n = np.linalg.norm(v)

```

```

if n < 1e-12:
    raise RuntimeError("Vector de cámara con norma cero.")

return v / n

def build_camera_basis(eye, target):
    eye = np.asarray(eye, dtype=np.float64)
    target = np.asarray(target, dtype=np.float64)

    forward = normalize_vector(target - eye)

    world_up = np.array([0.0, 1.0, 0.0], dtype=np.float64)

    right = np.cross(forward, world_up)

    if np.linalg.norm(right) < 1e-9:
        world_up = np.array([0.0, 0.0, 1.0], dtype=np.float64)
        right = np.cross(forward, world_up)

    right = normalize_vector(right)
    up = normalize_vector(np.cross(right, forward))

    return right, up, forward

def get_render_camera(config):
    eye = np.array([
        float(config["render_camera_eye_x"]),
        float(config["render_camera_eye_y"]),
        float(config["render_camera_eye_z"]),
    ], dtype=np.float64)

    target = np.array([
        float(config["render_camera_target_x"]),
        float(config["render_camera_target_y"]),
        float(config["render_camera_target_z"]),
    ], dtype=np.float64)

```

```

right, up, forward = build_camera_basis(eye, target)

return eye, right, up, forward

def make_red_near_colors(points, config):
    depth = points[:, 2]

    dmin = float(config["red_depth_min_m"])
    dmax = float(config["red_depth_max_m"])

    t = (depth - dmin) / max(dmax - dmin, 1e-6)
    t = np.clip(t, 0.0, 1.0)

    intensity = 1.0 - t

    r = 60.0 + 195.0 * intensity
    g = 0.0 + 35.0 * (1.0 - intensity)
    b = 0.0 + 35.0 * (1.0 - intensity)

    colors = np.stack([r, g, b], axis=1)
    colors = np.clip(colors, 0, 255).astype(np.uint8)

    return colors

def render_point_cloud_camera_view(
    points,
    colors_float_rgb,
    out_path,
    config,
    color_mode="rgb"
):
    if points.shape[0] == 0:
        raise RuntimeError("No hay puntos válidos para
renderizar.")

    width = int(config["render_width_px"])
    height = int(config["render_height_px"])

```

```

bg = np.array(config["render_background_rgb"], dtype=np.uint8)
canvas = np.zeros((height, width, 3), dtype=np.uint8)
canvas[:, :] = bg[None, None, :]

eye, right, up, forward = get_render_camera(config)

points64 = points.astype(np.float64)

valid_world = (
    np.isfinite(points64).all(axis=1)
    & (points64[:, 2] >= float(config["render_min_depth_m"]))
    & (points64[:, 2] <= float(config["render_max_depth_m"]))
    & (np.abs(points64[:, 0]) <=
float(config["render_max_abs_x_m"]))
    & (points64[:, 1] >= float(config["render_min_y_m"]))
    & (points64[:, 1] <= float(config["render_max_y_m"]))
)

points64 = points64[valid_world]
colors_float_rgb = colors_float_rgb[valid_world]

if points64.shape[0] == 0:
    raise RuntimeError("No quedan puntos válidos después del
recorte de render.")

rel = points64 - eye[None, :]

x_cam = rel @ right
y_cam = rel @ up
z_cam = rel @ forward

valid_cam = z_cam > 0.2

x_cam = x_cam[valid_cam]
y_cam = y_cam[valid_cam]
z_cam = z_cam[valid_cam]
points_used = points64[valid_cam]
colors_used = colors_float_rgb[valid_cam]

if x_cam.size == 0:

```



```

        raise RuntimeError("No quedan puntos delante de la cámara
virtual.")

    fov_rad = np.deg2rad(float(config["render_fov_deg"]))
    focal = width / (2.0 * np.tan(fov_rad / 2.0))

    u = width * 0.5 + focal * (x_cam / z_cam)
    v = height * 0.5 - focal * (y_cam / z_cam) +
float(config["render_principal_y_offset_px"])

    u_int = np.round(u).astype(np.int32)
    v_int = np.round(v).astype(np.int32)

    valid_img = (
        (u_int >= 0)
        & (u_int < width)
        & (v_int >= 0)
        & (v_int < height)
    )

    u_int = u_int[valid_img]
    v_int = v_int[valid_img]
    z_cam = z_cam[valid_img]
    points_used = points_used[valid_img]
    colors_used = colors_used[valid_img]

    if u_int.size == 0:
        raise RuntimeError("Ningún punto cae dentro del encuadre de
la cámara virtual.")

    if color_mode == "rgb":
        render_colors = np.clip(colors_used * 255.0, 0,
255).astype(np.uint8)

    elif color_mode == "red_near":
        render_colors = make_red_near_colors(points_used, config)

    else:
        raise ValueError(f"color_mode no soportado: {color_mode}")

```

```

order = np.argsort(z_cam)[::-1]

radius = int(config["render_point_radius"])

if radius <= 0:
    canvas[v_int[order], u_int[order]] = render_colors[order]
else:
    for idx in order:
        color = tuple(int(c) for c in render_colors[idx])
        cv2.circle(
            canvas,
            (int(u_int[idx]), int(v_int[idx])),
            radius,
            color,
            thickness=-1,
            lineType=cv2.LINE_AA
        )

Image.fromarray(canvas).save(out_path)

# =====
# 15. GUARDADO DE IMÁGENES DE ENTRADA Miguel Hernández
# =====

def copy_input_images(left_path, right_path, out_dir):
    left_img = Image.open(left_path).convert("RGB")
    right_img = Image.open(right_path).convert("RGB")

    left_out = os.path.join(out_dir, "imagen_izquierda.png")
    right_out = os.path.join(out_dir, "imagen_derecha.png")

    left_img.save(left_out)
    right_img.save(right_out)

    return left_out, right_out

def read_rgb_image_numpy(path):
    img = Image.open(path).convert("RGB")

```

```

return np.array(img, dtype=np.uint8)

def delete_deprecated_outputs(out_dir):
    """
    Elimina archivos antiguos que ya no se generan en esta versión.

    Esto evita que queden resultados de ejecuciones anteriores
    dentro de la
    carpeta de cada muestra y parezca que el código los ha creado
    de nuevo.
    """

    deprecated = [
        "vista_3d_estatica.png",
        "vista_3d_estatica_sin_ejes.png",
        "disparidad_color.png",
        "vista_3d_estatica_rojo_cerca.png",
    ]

    for filename in deprecated:
        path = os.path.join(out_dir, filename)

        if os.path.exists(path):
            os.remove(path)
            print(f"Archivo antiguo eliminado: {path}")

def resize_image_to_width(img, target_width):
    """
    Redimensiona una imagen manteniendo la relación de aspecto.
    """

    if img.width == target_width:
        return img

    scale = target_width / float(img.width)
    target_height = int(round(img.height * scale))

    if hasattr(Image, "Resampling"):

```

```

        resample = Image.Resampling.LANCZOS
    else:
        resample = Image.LANCZOS

    return img.resize((target_width, target_height),
resample=resample)

def save_vertical_summary_image(image_paths, out_path,
border_px=4):
    """
    Combina varias imágenes en vertical con recuadros negros
    límites.

    Características:
    - El orden de las imágenes viene dado por image_paths.
    - Todas las imágenes se ajustan al mismo ancho.
    - Cada imagen queda contenida en un recuadro negro.
    - Los recuadros son límites: la línea inferior de una
imagen es
    la misma línea superior de la imagen siguiente.
    - No se añaden espacios vacíos ni blancos entre imagen y
recuadro.
    """

    if len(image_paths) == 0:
        raise RuntimeError("No se recibieron imágenes para crear el
resumen vertical.")

    images = []

    for path in image_paths:
        if not os.path.exists(path):
            raise FileNotFoundError(f"No existe la imagen para el
resumen vertical: {path}")

        img = Image.open(path).convert("RGB")
        images.append(img)

    target_width = max(img.width for img in images)

```

```

    images = [resize_image_to_width(img, target_width) for img in
images]

border_px = int(border_px)

if border_px < 1:
    raise ValueError("border_px debe ser al menos 1.")

canvas_width = target_width + 2 * border_px
canvas_height = sum(img.height for img in images) + border_px *
(len(images) + 1)

canvas = Image.new("RGB", (canvas_width, canvas_height),
color=(0, 0, 0))

y = border_px

for img in images:
    canvas.paste(img, (border_px, y))
    y += img.height + border_px

canvas.save(out_path)

return out_path

# =====
# 16. PROCESAR UNA MUESTRA
# =====

def process_one_sample(
    sample,
    model,
    preprocess_module,
    device,
    config,
    checkpoint_path
):
    sample_name = sample["name"]
    sample_stem = sample["stem"]

```

```

    output_sample_dir = os.path.join(config["output_dir"],
sample_stem)
    ensure_dir(output_sample_dir)

    # Elimina vistas antiguas si existían de ejecuciones
anteriores.
    delete_deprecated_outputs(output_sample_dir)

print("\n=====
==")
    print(f"Procesando muestra: {sample_name}")

print("=====
")
    print(f"Imagen izquierda: {sample['left_path']}")
    print(f"Imagen derecha: {sample['right_path']}")
    print(f"Calibración: {sample['calib_path']}")
    print(f"Salida: {output_sample_dir}")

    left_saved, right_saved = copy_input_images(
        sample["left_path"],
        sample["right_path"],
        output_sample_dir
    )

    calib_params =
get_camera_params_from_kitti_calib(sample["calib_path"])

    print(
        "Calibración usada: "
        f"fx={calib_params['fx']:.6f}, "
        f"fy={calib_params['fy']:.6f}, "
        f"cx={calib_params['cx']:.6f}, "
        f"cy={calib_params['cy']:.6f}, "
        f"baseline={calib_params['baseline']:.6f} m"
    )

    pred_disp, inference_time, pad_info = predict_disparity_psmnet(

```

```

        model=model,
        preprocess_module=preprocess_module,
        left_path=sample["left_path"],
        right_path=sample["right_path"],
        device=device
    )

    print(f"Tiempo de inferencia PSMNet: {inference_time:.4f} s")
    print(f"Forma disparidad predicha: {pred_disp.shape}")
    print(
        f"Rango disparidad: min={np.nanmin(pred_disp):.4f}, "
        f"max={np.nanmax(pred_disp):.4f}, "
        f"media={np.nanmean(pred_disp):.4f}"
    )

    # -----
    # Guardar disparidad
    # -----

    disp_npy_path = os.path.join(output_sample_dir,
    "disparidad_predicha.npy")
    np.save(disp_npy_path, pred_disp)

    disp_colorbar_path = os.path.join(output_sample_dir,
    "disparidad_predicha_con_barra.png")
    save_disparity_with_colorbar(
        disp=pred_disp,
        out_path=disp_colorbar_path,
        config=config,
        title="Predicción de disparidad"
    )

    # -----
    # Nube de puntos y profundidad
    # -----

    rgb_left = read_rgb_image_numpy(sample["left_path"])

    points_ply, colors_ply, depth, valid_cloud_mask =
    make_point_cloud_from_disparity(

```

```

        disp=pred_disp,
        rgb_image=rgb_left,
        calib_params=calib_params,
        config=config,
        point_stride=config["point_stride_ply"]
    )

    print(f"Puntos válidos para PLY: {points_ply.shape[0]}")

    depth_npy_path = os.path.join(output_sample_dir,
    "profundidad_metros.npy")
    np.save(depth_npy_path, depth)

    depth_color_path = os.path.join(output_sample_dir,
    "profundidad_color.png")
    save_depth_color(
        depth=depth,
        out_path=depth_color_path,
        config=config,
        title="Estimación de profundidad"
    )

    ply_path = os.path.join(output_sample_dir,
    "nube_puntos_color.ply")
    save_point_cloud_ply(
        points=points_ply,
        colors=colors_ply,
        out_path=ply_path
    )

    # -----
    # HTML interactivo
    # -----

    points_html, colors_html, _, _ =
    make_point_cloud_from_disparity(
        disp=pred_disp,
        rgb_image=rgb_left,
        calib_params=calib_params,
        config=config,

```



```

        point_stride=config["point_stride_html"]
    )

    points_html, colors_html = sample_points_for_visualization(
        points_html,
        colors_html,
        max_points=config["max_points_html"],
        seed=config["random_seed"]
    )

    print(f"Puntos usados en HTML interactivo:
    {points_html.shape[0]}")

    html_path = os.path.join(output_sample_dir,
    "nube_puntos_interactiva.html")
    save_interactive_point_cloud_html(
        points=points_html,
        colors=colors_html,
        out_path=html_path,
        config=config,
        title=f"Nube de puntos interactiva - {sample_stem}"
    )

# -----
# Vistas estáticas renderizadas
# -----

    points_static, colors_static = sample_points_for_visualization(
        points_ply,
        colors_ply,
        max_points=config["max_points_static"],
        seed=config["random_seed"]
    )

    static_rgb_path = os.path.join(output_sample_dir,
    "vista_3d_estatica_rgb.png")
    render_point_cloud_camera_view(
        points=points_static,
        colors_float_rgb=colors_static,
        out_path=static_rgb_path,

```

```

        config=config,
        color_mode="rgb"
    )

    # -----
    # Imagen resumen vertical
    # -----

    summary_visual_path = os.path.join(output_sample_dir,
"resumen_visual_vertical.png")
    save_vertical_summary_image(
        image_paths=[
            left_saved,
            disp_colorbar_path,
            depth_color_path,
            static_rgb_path,
        ],
        out_path=summary_visual_path,
        border_px=4
    )

    # -----
    # Resumen JSON
    # -----

    valid_depth = (
        np.isfinite(depth)
        & (depth >= config["min_depth_m"])
        & (depth <= config["max_depth_m"])
    )

    valid_disp = np.isfinite(pred_disp) & (pred_disp >=
config["min_disp"])

    summary = {
        "sample": sample_name,
        "left_path_original": sample["left_path"],
        "right_path_original": sample["right_path"],
        "calib_path": sample["calib_path"],
        "output_dir": output_sample_dir,
    }

```

```

        "checkpoint_path": checkpoint_path,
        "maxdisp": config["maxdisp"],

        "image_shape_h_w": [int(pred_disp.shape[0]),
int(pred_disp.shape[1])],

        "calibration": {
            "left_projection_key":
calib_params["left_projection_key"],
            "right_projection_key":
calib_params["right_projection_key"],
            "fx": safe_float_for_json(calib_params["fx"]),
            "fy": safe_float_for_json(calib_params["fy"]),
            "cx": safe_float_for_json(calib_params["cx"]),
            "cy": safe_float_for_json(calib_params["cy"]),
            "baseline_m":
safe_float_for_json(calib_params["baseline"]),
        },

        "psmnet_padding": pad_info,

        "inference_time_sec": safe_float_for_json(inference_time),

        "disparity_stats": {
            "min": safe_float_for_json(np.nanmin(pred_disp)),
            "max": safe_float_for_json(np.nanmax(pred_disp)),
            "mean": safe_float_for_json(np.nanmean(pred_disp)),
            "valid_pixels_disp": int(valid_disp.sum()),
        },

        "depth_stats_m": {
            "min":
safe_float_for_json(np.nanmin(depth[valid_depth])) if
valid_depth.sum() > 0 else None,
            "max":
safe_float_for_json(np.nanmax(depth[valid_depth])) if
valid_depth.sum() > 0 else None,

```

```

        "mean":
safe_float_for_json(np.nanmean(depth[valid_depth])) if
valid_depth.sum() > 0 else None,
        "valid_pixels_depth": int(valid_depth.sum()),
        "min_depth_m_config":
safe_float_for_json(config["min_depth_m"]),
        "max_depth_m_config":
safe_float_for_json(config["max_depth_m"]),
    },

    "point_cloud": {
        "points_ply": int(points_ply.shape[0]),
        "points_html": int(points_html.shape[0]),
        "point_stride_ply": int(config["point_stride_ply"]),
        "point_stride_html": int(config["point_stride_html"]),
        "remove_top_ratio_for_cloud":
safe_float_for_json(config["remove_top_ratio_for_cloud"]),
        "flip_x_for_visual_match":
bool(config["flip_x_for_visual_match"]),
    },

    "render_static_view": {
        "render_width_px": config["render_width_px"],
        "render_height_px": config["render_height_px"],
        "render_fov_deg": config["render_fov_deg"],
        "render_camera_eye": [
            config["render_camera_eye_x"],
            config["render_camera_eye_y"],
            config["render_camera_eye_z"],
        ],
        "render_camera_target": [
            config["render_camera_target_x"],
            config["render_camera_target_y"],
            config["render_camera_target_z"],
        ],
        "render_min_depth_m": config["render_min_depth_m"],
        "render_max_depth_m": config["render_max_depth_m"],
        "render_point_radius": config["render_point_radius"],
    },
},

```

```

        "generated_files": {
            "imagen_izquierda": left_saved,
            "imagen_derecha": right_saved,
            "disparidad_predicha_npy": disp_npy_path,
            "disparidad_predicha_con_barra_png":
disp_colorbar_path,
            "profundidad_metros_npy": depth_npy_path,
            "profundidad_color_png": depth_color_path,
            "nube_puntos_color_ply": ply_path,
            "nube_puntos_interactiva_html": html_path,
            "vista_3d_estatica_rgb_png": static_rgb_path,
            "resumen_visual_vertical_png": summary_visual_path,
        }
    }

    summary_path = os.path.join(output_sample_dir,
"resumen_reconstruccion.json")

    with open(summary_path, "w") as f:
        json.dump(summary, f, indent=2)

    print("Archivos generados:")
    print(f"  Disparidad con barra: {disp_colorbar_path}")
    print(f"  Profundidad color: {depth_color_path}")
    print(f"  Nube PLY: {ply_path}")
    print(f"  Nube HTML interactiva: {html_path}")
    print(f"  Vista estática RGB: {static_rgb_path}")
    print(f"  Resumen visual vertical:
{summary_visual_path}")
    print(f"  Resumen JSON: {summary_path}")

    return summary

# =====
# 17. EJECUCIÓN COMPLETA
# =====

def run_reconstruction_kitti2015_test(config):
    assert_gpu_available()

```

```

print_gpu_info()

ensure_dir(config["reconstruction_base_dir"])
ensure_dir(config["output_dir"])

checkpoint_path = get_selected_checkpoint_path(config)

config_to_save = dict(config)
config_to_save["checkpoint_path_selected"] = checkpoint_path

global_config_path = os.path.join(config["output_dir"],
"reconstruction_config.json")

with open(global_config_path, "w") as f:
    json.dump(config_to_save, f, indent=2)

print("Configuración guardada en:")
print(global_config_path)

kitti_testing_root, calib_testing_root =
prepare_kitti2015_test(config)

samples = build_kitti2015_test_pairs(
    kitti_testing_root=kitti_testing_root,
    calib_testing_root=calib_testing_root,
    config=config
)

stackhourglass, preprocess_module, repo_dir =
import_psmnet(config)

device = torch.device("cuda")

print("Creando modelo PSMNet...")
model = stackhourglass(config["maxdisp"]).to(device)

model = load_checkpoint_into_model(
    model=model,
    ckpt_path=checkpoint_path,
    device=device

```

```

)

model.eval()

all_summaries = []

total_start = time.time()

for idx, sample in enumerate(samples):

print("\n#####  
##")
    print(f"MUESTRA {idx + 1}/{len(samples)}")

print("#####  
")

    summary = process_one_sample(
        sample=sample,
        model=model,
        preprocess_module=preprocess_module,
        device=device,
        config=config,
        checkpoint_path=checkpoint_path
    )

    all_summaries.append(summary)

total_time = time.time() - total_start

global_summary = {
    "num_samples": len(all_summaries),
    "total_time_sec": safe_float_for_json(total_time),
    "mean_time_per_sample_sec": safe_float_for_json(total_time
/ len(all_summaries)) if len(all_summaries) > 0 else None,
    "reconstruction_base_dir":
config["reconstruction_base_dir"],
    "weights_dir": config["weights_dir"],
    "output_dir": config["output_dir"],
    "checkpoint_path": checkpoint_path,

```

```

        "psmnet_repo": repo_dir,
        "kitti_testing_root": kitti_testing_root,
        "calib_testing_root": calib_testing_root,
        "selected_test_filenames":
config["selected_test_filenames"],
        "samples": all_summaries,
    }

    global_summary_path = os.path.join(
        config["output_dir"],
        "resumen_global_reconstruccion.json"
    )

    with open(global_summary_path, "w") as f:
        json.dump(global_summary, f, indent=2)

print("\n=====
==")
    print("RECONSTRUCCIÓN FINALIZADA")

print("=====
")
    print(f"Número de muestras procesadas: {len(all_summaries)}")
    print(f"Tiempo total: {total_time:.2f} s")
    print(f"Carpeta base: {config['reconstruction_base_dir']}")
    print(f"Pesos usados: {checkpoint_path}")
    print(f"Salida general: {config['output_dir']}")
    print(f"Resumen global: {global_summary_path}")

print("=====
")

    return global_summary

```



```
# =====  
# 18. LANZAR RECONSTRUCCIÓN  
# =====  
  
try:  
    resultado_reconstruccion =  
run_reconstruction_kitti2015_test(CONFIG)  
    resultado_reconstruccion  
  
finally:  
    print("Desconectando sesión de Colab...")  
    runtime.unassign()
```

