

UNIVERSIDAD DE VALLADOLID
MÁSTER UNIVERSITARIO
Ingeniería Informática



TRABAJO FIN DE MÁSTER

**Estimación de Profundidad Monocular Online con
Transformers Eficientes**

Realizado por **GUILLERMO SÁNCHEZ BRIZUELA**



Universidad de Valladolid

24 de mayo de 2022

Tutor: Aníbal Bregón Bregón

Universidad de Valladolid



Máster universitario en Ingeniería Informática

D. Anibal Bregón Bregón, profesor del departamento de Informática, área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. GUILLERMO SÁNCHEZ BRIZUELA, ha realizado el Trabajo final de Máster en Ingeniería Informática titulado "ESTIMACIÓN DE PROFUNDIDAD MONOCULAR ONLINE CON TRANSFORMERS EFICIENTES".

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Valladolid, 24 de mayo de 2022

Vº. Bº. del Tutor:

D. Anibal Bregón Bregón

Agradecimientos

Antes de comenzar esta memoria me gustaría agradecer a Anibal Bregón, mi tutor, el esfuerzo y el tiempo que ha dedicado para que este proyecto llegue a buen puerto, así como las direcciones y consejos que me ha dado durante su desarrollo.

Igualmente, gracias a mis padres por todo lo que me han dado a lo largo de mi vida; a mis amigos y familiares por apoyarme y animarme; y por supuesto, a Daniela por acompañarme en todo momento.

De corazón, gracias.

Resumen

La estimación de profundidad monocular consiste en recuperar automáticamente una aproximación de la dimensión perdida al proyectar una escena tridimensional en una imagen bidimensional. Este problema tiene infinitas soluciones geométricas, por lo que es prácticamente imposible resolverlo con técnicas de visión artificial tradicional. Sin embargo, las técnicas de Deep Learning son capaces de extraer distintas características de las imágenes que permiten aproximar una solución. En este trabajo se estudia este problema y las soluciones existentes, especialmente aquellas basadas en Transformers y aprendizaje supervisado. En una de estas soluciones, se llevan a cabo una serie de modificaciones y desarrollos que permiten reducir el tamaño del modelo original y multiplicar por cerca de cinco su velocidad de inferencia. Además, se incluye un estudio exhaustivo, tanto cuantitativo como cualitativo, de la influencia de las modificaciones evaluando los modelos en el conjunto de datos KITTI, orientado a conducción autónoma.

Descriptores

Estimación de Profundidad Monocular, Deep Learning, Transformers, Redes Neuronales Convolucionales, Visión Artificial

Abstract

Monocular depth estimation deals with the automatic recovery of an approximation of the dimension that is lost when projecting a three-dimensional scene into a two-dimensional image. This problem has an infinite number of geometric solutions, which makes it practically impossible to solve using traditional computer vision techniques. Nonetheless, Deep Learning techniques are capable of extracting different characteristics from the images that make it possible to approximate a solution. In this work this problem and the existing solutions are studied, especially those based on Transformers and supervised learning. In one of these solutions, a series of modifications and developments are carried out to reduce the size of the original model and multiply its inference speed by nearly five. Furthermore, an exhaustive study, both quantitative and qualitative, of the influence of the different modifications is included, evaluating the models in the KITTI dataset, oriented to autonomous driving.

Keywords

Monocular Depth Estimation, Deep Learning, Transformers, Convolutional Neural Networks, Computer Vision

Índice general

Índice general	III
Índice de figuras	VI
Índice de tablas	VIII
1. Introducción	1
1.1. Motivación	2
1.2. Objetivos	3
1.3. Estructura del documento	3
2. Marco Teórico y Estado del Arte	4
2.1. Inteligencia Artificial, Aprendizaje automático, y Aprendizaje Profundo	4
2.2. Redes neuronales	4
2.2.1. Funciones de activación	6
2.2.2. Funciones de pérdida	6
2.2.3. Optimizadores	7
2.3. MLP	9
2.4. RNN	9
2.5. CNN	10
2.5.1. Conexiones residuales y ResNets	12
2.5.2. EfficientNet	13
2.6. Transformers	14
2.6.1. Arquitectura	14
2.6.2. Mecanismos de atención	15
2.6.3. Atención eficiente	16
2.6.3.1. Patrones fijos - Fixed Patterns (FP)	16
2.6.3.2. Combinación de patrones - Combination of Patterns (CP)	16
2.6.3.3. Aprendizaje de patrones - Learnable Patterns (LP)	16
2.6.3.4. Disminución de rango	17
2.6.4. Performer	17
2.7. Transformers para visión artificial - ViT	18
2.8. Estimación de profundidades	19
2.8.1. Técnicas de estimación de profundidades	19
2.8.2. Aprendizaje automático no supervisado	20
2.8.3. Aprendizaje automático semisupervisado	20
2.8.4. Aprendizaje automático supervisado	21

2.8.5. DPT	21
2.9. Optimización de modelos	23
3. Material y Métodos	27
3.1. Software	27
3.2. Hardware	29
3.3. Datasets	30
3.3.1. ImageNet	30
3.3.2. MIX6	31
3.3.3. KITTI	31
3.3.3.1. Datos	32
3.3.3.2. Conjuntos de entrenamiento, validación y evaluación	33
3.4. Definición de modelos a entrenar	34
3.5. Evaluación	34
3.5.1. Métricas	34
3.5.1.1. <i>Accuracy under a threshold</i>	34
3.5.1.2. <i>Mean Absolute Value of the Relative Error (Abs Rel)</i>	35
3.5.1.3. <i>Mean Squared Relative Error (Sq Rel)</i>	35
3.5.1.4. <i>Linear Root Mean Squared Error (RMSE)</i>	35
3.5.1.5. <i>Logarithmic Root Mean Squared Error (RMSElog)</i>	35
3.5.1.6. <i>Scale Invariant Logarithmic Error (SIlog)</i>	36
3.5.1.7. <i>Mean Logarithmic Error (Log10)</i>	36
3.5.1.8. Velocidad de procesamiento	36
4. Desarrollo y Modificaciones de la Arquitectura	37
4.1. Cloud	37
4.2. Warmstart	38
4.3. Entrenamiento	39
4.3.1. Función de pérdida	40
4.4. Modificaciones introducidas en DPT	41
4.4.1. Reducción de tamaño de la entrada	41
4.4.2. Capas de atención eficiente	41
4.4.3. Número de cabezas	41
4.4.4. Hooks del transformer y bloques de atención posteriores	42
4.4.5. Backbone convolucional	43
4.5. Pruebas de cuantificación	44
5. Resultados	45
5.1. Resultados cuantitativos	45
5.1.1. Reducción del tamaño de la entrada	45
5.1.2. Cambio del backbone convolucional	46
5.1.3. Número de cabezas	48
5.1.4. Capas de atención eficiente	49
5.1.5. Cambio en los hooks del transformer y eliminación de las capas de atención posteriores	50
5.2. Tamaño de los modelos y número de operaciones	51
5.3. Resultados cualitativos	52
6. Discusión	57
7. Conclusiones y Lineas Futuras	60

Apéndices	61
Apéndice A Planificación del proyecto	62
Apéndice B Documentación	63
B.1. Introducción	63
B.2. Estructura de directorios	63
B.3. Descarga de datos e instalación de <i>software</i>	64
B.3.1. Descarga de datos	65
B.3.2. Docker Engine	65
B.3.3. NVIDIA Container Toolkit	66
B.3.4. Construcción de la imagen de Docker	67
B.3.5. Instalación de wandb (Opcional)	67
B.4. Ejecución	68
B.5. Despliegue en la nube	69
Apéndice C Demostración de la independencia de la escala	71
Apéndice D Ficheros de resultados	72
Bibliografía	73

Índice de figuras

1.1.	Proyección perspectiva y ambigüedad de escala.	1
2.1.	Estructura de una neurona artificial.	5
2.2.	Funciones de activación.	6
2.3.	Perceptrón Multicapa.	9
2.4.	Red recurrente.	10
2.5.	Operación de convolución con un <i>kernel</i> de 3×3 y un <i>stride</i> de 1 para uno de los valores de salida.	10
2.6.	Operación de Pooling2D.	11
2.7.	Primeros pasos de una convolución transpuesta.	11
2.8.	<i>Pointwise convolution</i>	12
2.9.	<i>Depthwise convolution</i>	12
2.10.	Conexión residual básica.	13
2.11.	Arquitectura del <i>transformer</i> . Fuente: [1]	14
2.12.	Producto escalar para el cálculo de la atención (izquierda) y bloque de <i>Multi-Head Attention</i> (derecha). Fuente: [1]	16
2.13.	Aproximación y reasociación del orden de multiplicación en el mecanismo de atención del <i>Performer</i> . Figura inspirada en [2].	17
2.14.	<i>Backbone</i> basado en fragmentos y proyección lineal del ViT. Figura inspirada en [3]	18
2.15.	<i>Backbone</i> convolucional del ViT-Hybrid.	19
2.16.	Esquema de la arquitectura de DPT-Hybrid.	22
2.17.	Bloques base del <i>decoder</i> de DPT. Izquierda: Bloque de <i>Reassemble</i> . Derecha: Bloque de fusión. Figura adaptada de [4].	23
3.1.	Interfaz web de <i>Weights and Biases</i>	28
3.2.	Muestra de las imágenes de ImageNet. Fuente: [5]	30
3.3.	Muestra de las cuatro imágenes en bruto disponibles en KITTI para un instante dado.	32
3.4.	Anotación de KITTI y detalle de su carácter disperso para un instante dado.	33
4.1.	Esquema de la configuración llevada a cabo en la nube.	38
4.2.	Cambio en los <i>hooks</i> y en el número de bloques de atención.	43
5.1.	FPS promedios de los modelos en función de la precisión durante la inferencia. Las barras grises representan la desviación estándar de las medidas.	45
5.2.	FPS medios de los modelos en función del tamaño de su entrada. Las barras grises representan la desviación estándar de las medidas.	45
5.3.	FPS medios de los modelos en función del <i>backbone</i> utilizado y del tamaño de la entrada. Las barras grises representan la desviación estándar de las medidas.	46
5.4.	SIlog obtenido en el conjunto de validación para cada uno de los 36 modelos entrenados.	47
	Más bajo es mejor.	

5.5. Comparación de las curvas de aprendizaje del SIlog durante el entrenamiento y validación de dos modelos con diferentes <i>backbones</i> e inicializaciones de pesos. Valores suavizados con una media exponencial ponderada para facilitar su visualización.	47
5.6. Comparación del SIlog en el conjunto de validación durante el entrenamiento en tres modelos con diferentes inicializaciones de pesos.	48
5.7. SIlog obtenido en el conjunto de validación para cada uno de los 36 modelos entrenados con distintos ejes en función del <i>backbone</i> . Más bajo es mejor.	48
5.8. FPS medios de los modelos en función del número de cabezas utilizado y del tamaño de la entrada. Las barras grises representan la desviación estándar de las medidas.	49
5.9. FPS medios de los modelos en función del mecanismo de atención utilizado y del tamaño de la entrada. Las barras grises representan la desviación estándar de las medidas.	49
5.10. Comparación de la complejidad y el tiempo de inferencia en un bloque de atención usando el mecanismo de atención estándar y el mecanismo eficiente del <i>Performer</i> . Detalle de los tiempos de inferencia para entradas con un número de <i>tokens</i> reducido a la izquierda.	50
5.11. FPS medios de los modelos en función de los <i>hooks</i> utilizados y del tamaño de la entrada. Las barras grises representan la desviación estándar de las medidas.	50
5.12. Tamaño en memoria de los modelos en función de su configuración.	51
5.13. Número de operaciones en coma flotante necesarias para inferir la profundidad en una imagen en función de la configuración del modelo.	51
5.14. Resultados en función de la resolución de entrada. (a) Entrada del modelo y detalle. (b) Salida del modelo proporcionado en el repositorio de DPT [4] (sin reducir la entrada). (c) Salida del modelo equivalente a DPT con el tamaño de entrada reducido.	53
5.15. Resultados en función del <i>backbone</i> empleado. (a) Entrada del modelo y detalle. (b) Salida del modelo equivalente a DPT con el tamaño de entrada reducido (ResNet50). (c) Salida del modelo equivalente a DPT con el tamaño de entrada reducido y EfficientNet-B0 como <i>backbone</i>	53
5.16. Resultados en función del número de cabezas de atención del modelo. (a) Entrada del modelo y detalle. (b) Salida del modelo equivalente a DPT con el tamaño de entrada reducido (12 cabezas de atención). (c) Salida del modelo equivalente a DPT con el tamaño de entrada reducido y 1 cabeza de atención. (d) Salida del modelo equivalente a DPT con el tamaño de entrada reducido y 24 cabezas de atención.	54
5.17. Resultados en función de los <i>hooks</i> empleados. (a) Entrada del modelo y detalle. (b) Salida del modelo equivalente a DPT con el tamaño de entrada reducido (atención estándar). (c) Salida del modelo equivalente a DPT con el tamaño de entrada reducido y atención tipo <i>Performer</i>	55
5.18. Resultados en función de los <i>hooks</i> empleados. (a) Entrada del modelo y detalle. (b) Salida del modelo equivalente a DPT con el tamaño de entrada reducido (<i>hooks</i> en bloques [8, 11]). (c) Salida del modelo equivalente a DPT con el tamaño de entrada reducido y <i>hooks</i> en los bloques [2, 5]. (d) Salida del modelo equivalente a DPT con el tamaño de entrada reducido y <i>hooks</i> en los bloques [0, 1].	56
A.1. Cronograma del proyecto.	62
B.1. Resultado del comando nvidia-smi dentro del contenedor.	67

Índice de tablas

3.1.	Especificaciones de los equipos empleados durante el trabajo de fin de máster.	29
3.2.	Datasets que conforman MIX6. Subrayados aquellos que no forman parte de MIX5.. .	31
3.3.	Distribución de las imágenes y número de imágenes no encontradas en el dataset. . .	33
6.1.	Comparación de distintos modelos del estado del arte. Resultados medidos en el conjunto de test del <i>Eigen Split</i> de KITTI. Tiempos de inferencia calculados con precisión mixta en el Equipo 1 de la Tabla 3.1. Se muestran los incrementos/decrementos de las métricas de las modificaciones de DPT-Hybrid respecto de la arquitectura original.	58

1: Introducción

Cuando usamos una cámara para capturar una imagen o un vídeo, creamos una representación bidimensional de lo que es en realidad una escena tridimensional. Para conseguir esta reducción de dimensiones, se proyectan en un plano, cada uno de los puntos visibles. Al realizar esta proyección, se pierde una gran cantidad de información relacionada con la profundidad. Esto es debido a que los puntos ahora representados en el plano bidimensional podían encontrarse a cualquier distancia siempre y cuando estuviesen situados en la recta que atraviesa el punto verdadero y el centro de la cámara ([Figura 1.1](#)). Es decir, existen infinitas escenas tridimensionales con la misma proyección perspectiva.

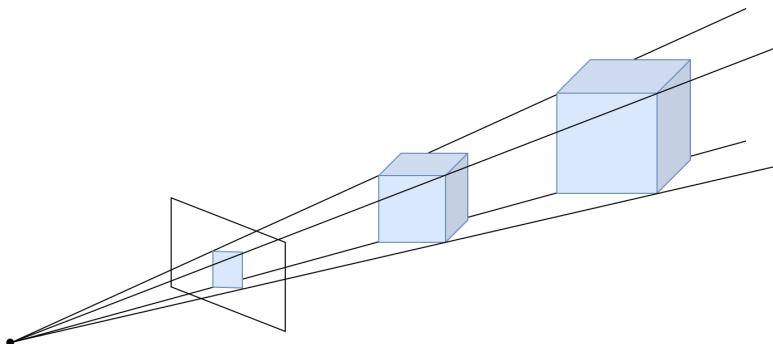


Figura 1.1: Proyección perspectiva y ambigüedad de escala.

Existen soluciones *hardware* capaces de capturar escenas tridimensionales sin esta perdida de información, por ejemplo: sensores LiDAR, cámaras de tiempo de vuelo, o conjuntos de cámaras para llevar a cabo estereovisión¹[\[6\]](#). Sin embargo, estas soluciones requieren de sensores adicionales, con el incremento de material, coste y peso que esto conlleva. Es por esto por lo que recuperar la profundidad (o una estimación de ésta) a partir de una imagen obtenida con una cámara corriente sería de gran utilidad, por ejemplo:

- En diferentes tareas dentro de múltiples campos de aplicación: Navegación robótica y conducción autónoma, empleando la profundidad para reconstruir mapas o calcular el cambio en la posición del agente (odometría visual, VSLAM); detección de superficies y capacidad de incluir ocultación en aplicaciones de realidad aumentada; generación de modelos 3D a través de fotografías (fotogrametría); efectos fotográficos en aplicaciones móviles (efecto *Bokeh*); etc.

¹Con una pareja de cámaras de posiciones conocidas, es posible estimar la profundidad a partir de la disparidad geométrica entre las dos imágenes capturadas.

- Como información adicional o etapa intermedia en otras tareas típicas en visión artificial: detección de objetos, segmentación, clasificación, etc.

Si observamos, nunca mejor dicho, el sistema visual de los humanos, podemos comprobar como este es un sistema estereoscópico compuesto por dos “cámaras” (los ojos) y un cerebro que interpreta la disparidad entre estas imágenes para obtener una estimación de la profundidad a la que se encuentra cada objeto que vemos. No obstante, si nos tapamos un ojo, aunque con peores resultados, somos capaces de estimar la distancia a la que se encuentran los elementos que están dentro de nuestro campo visual, manteniendo, en una gran mayoría de las ocasiones, la capacidad de distinguir cuáles están más cerca y cuáles más lejos. Esto se debe en gran parte a una serie de señales que los humanos aprendemos a medida que crecemos, conocidos como pistas monoculares (pueden ser dinámicas o estáticas, en función de si consideran la información a lo largo del tiempo, por ejemplo, objetos en movimiento), y que no solo se emplean cuando nos tapamos un ojo, si no que también los emplea el cerebro cuando vemos con los dos ojos. Algunas de las principales pistas monoculares (estáticas) [7] son: el tamaño relativo con el que observamos un objeto en función de la distancia a la que se encuentre; la oclusión de los elementos que están más próximos que otros; la convergencia de líneas paralelas a medida que se alejan, por ejemplo, en carreteras o vías; el cambio en el tono del color de los objetos lejanos debido a la dispersión de la luz; o la forma de los reflejos y las sombras que producen los elementos, originados por las fuentes de luz de la escena.

No obstante, realizar este análisis de las imágenes monoculares, que tan eficientemente llevamos a cabo los humanos, en un ordenador de forma automática empleando técnicas de visión artificial tradicional roza lo imposible.

Las limitaciones de este tipo de técnicas no solo aparecen en la estimación de profundidades, sino que están presentes en una gran parte de las tareas propias de la visión artificial. Buscando dejar atrás estas limitaciones, en los últimos años se han desarrollado un gran número de sistemas de aprendizaje automático profundo (*Deep Learning*) diseñados para trabajar con imágenes [8, 9, 10]. Estos sistemas, pese a tener ciertos inconvenientes, han conseguido ofrecer unos muy buenos resultados junto a una mucho mayor robustez y capacidad de generalización ante modificaciones en las entradas (cambios de entorno, color, luz, orientación de los elementos, etc.), muy frecuentes en los entornos reales no controlados.

Dentro de estas técnicas, en general, y especialmente para la estimación de profundidades, las redes neuronales convolucionales han prevalecido como las arquitecturas que mejores resultados aportaban [11, 12, 13, 14]. No obstante, en los últimos años han surgido otro tipo de arquitecturas, los *transformers* [1], que presentan resultados que igualan e incluso superan aquellos obtenidos con redes convolucionales. En vista de esto, este trabajo revisa el estado del arte actual en estimación de profundidad monocular y explora una de las arquitecturas que mejores resultados consigue, DPT (Dense Prediction Transformers) [4], reproduciendo su entrenamiento y modificándola para disminuir su tamaño y acelerar la inferencia de resultados, es decir, reducir el tiempo necesario para estimar la profundidad en una imagen dada.

1.1. Motivación

Este trabajo tiene dos motivaciones claramente diferenciadas, la primera, académica y la segunda, más aplicada. En primer lugar, para poder trabajar en el campo de la estimación de profundidades, es necesario explorar y conocer las distintas técnicas que han surgido a lo largo de los años, tanto las bases y los primeros enfoques para resolver el problema, como las soluciones especializadas que conforman el estado del arte. Por lo tanto, en este trabajo se pretende resumir y agrupar las principales técnicas para poder, posteriormente, revisar y estudiar los temas que se tratan de una manera dirigida y ordenada.

Por otro lado, los modelos del estado del arte tienden a ser (con excepciones) cada vez más complejos, tienen un mayor número de parámetros, y precisan de grandes cantidades de datos para ser entrenados. Esto conlleva una perdida de accesibilidad al desarrollo y experimentación con dichas arquitecturas, que necesitan una infraestructura costosa para ejecutarse. Además, este incremento en tamaño de los modelos hace que sus velocidades de ejecución e inferencia de resultados se vea afectada. En muchas de las aplicaciones mencionadas en el apartado anterior, el tiempo de inferencia es un factor crítico, ya que muchas veces el procesamiento de las imágenes debe llevarse a cabo en entornos con recursos computacionales limitados y de forma *online*, es decir, procesar las imágenes a medida que están disponibles (sin considerar las restricciones de un entorno de tiempo real). En el caso de que la inferencia de los modelos no se lleve a cabo en dispositivos embebidos y recaiga en servidores a los que los clientes hacen peticiones, un mayor tiempo de ejecución se traduce directamente en un incremento de costes, por lo que tampoco es despreciable. Por estas razones, en este trabajo se busca modificar una de los modelos del estado del arte en estimación de profundidades a partir de imágenes monoculares para reducir su tamaño y tiempo de inferencia reduciendo lo mínimo posible la calidad de los resultados.

1.2. Objetivos

Los objetivos principales de este Trabajo Fin de Máster, son:

1. Llevar a cabo una revisión del estado del arte relacionado con la estimación de profundidades en imágenes monoculares. Más concretamente, en aquellas técnicas que empleen aprendizaje automático, prestando especial atención a las arquitecturas basadas en *transformers* y sus variantes eficientes.
2. Estudiar una arquitectura del estado del arte de estimación de profundidades y modificar su estructura para acelerar la inferencia hasta obtener modelos capaces de procesar imágenes de forma *online*. Además, una vez definidos los modelos con las variaciones planteadas, comparar sus resultados tras ajustarlos a un *dataset* concreto.
3. Explorar diferentes técnicas generales para acelerar el entrenamiento e inferencia de los modelos de aprendizaje automático profundo.
4. Diseñar una solución en la nube que permita desplegar de forma automática instancias que ejecuten los experimentos necesarios, es decir, entrenando los distintos modelos planteados.

1.3. Estructura del documento

Este trabajo Fin de Máster está organizado de la siguiente manera: Primero, se han expuesto en esta sección de Introducción tanto la motivación detrás del proyecto como los objetivos planteados; a continuación, en el segundo capítulo, se contextualiza el trabajo repasando las bases teóricas sobre las que se apoya su desarrollo y el Estado del Arte de estos campos; en el tercer capítulo, se presentan y justifican los materiales empleados para el desarrollo del trabajo y la metodología que se ha seguido; después, en el cuarto capítulo, se definen y explican aquellos desarrollos especialmente significativos dentro del trabajo para continuar en el quinto capítulo con los resultados que se han obtenido, haciendo especial hincapié en la influencia en estos resultados de cada uno de los desarrollos llevados a cabo. En el capítulo seis, se incluye una discusión de los resultados, y en el capítulo siete, las conclusiones del documento junto a una serie de líneas de investigación futuras que podrían explorarse para continuar trabajando en el contexto de este proyecto. Al final de este documento, se encuentra una sección de Apéndices con información y material complementario.

2: Marco Teórico y Estado del Arte

2.1. Inteligencia Artificial, Aprendizaje automático, y Aprendizaje Profundo

La Inteligencia Artificial (IA), engloba el estudio de agentes que perciben un entorno y actúan en consecuencia. Este campo, comprende multitud de disciplinas y técnicas aplicables en incontables escenarios. Dentro de los subcampos que conforman la IA están, por ejemplo, los sistemas expertos, los modelos probabilísticos, la lógica difusa, o, de especial importancia en este trabajo, el aprendizaje automático (*Machine Learning - ML*).

El aprendizaje automático estudia diferentes modelos que son capaces de extraer información de forma automática a partir de un conjunto de datos. En función de qué información tengan estos datos y qué se busque hacer con ella, es posible distinguir entre **aprendizaje supervisado** [15] (donde los datos están etiquetados con lo que se quiere aprender), **aprendizaje no supervisado** [16] (donde los datos no tienen anotaciones y se busca extraer información o encontrar relaciones entre ellos), **aprendizaje semisupervisado** [17] (si se usan datos etiquetados y no etiquetados) o **aprendizaje por refuerzo** [18] (si se emplea una función de recompensa y se deja al modelo elegir las acciones que lleva a cabo).

Este proyecto, no obstante, se centra más aún en el aprendizaje profundo (*Deep Learning - DL*), un subcampo del aprendizaje automático donde se emplean modelos más complejos, no lineales y con un número de parámetros a entrenar mucho mayor, que requieren de una cantidad de datos acorde para poder aprender de forma satisfactoria.

Cabe mencionar, que pese a que muchos de los conceptos explicados a continuación son perfectamente validos en otras modalidades, al centrar este trabajo en el aprendizaje profundo supervisado, el marco teórico se ajusta y limita a este.

2.2. Redes neuronales

Las redes neuronales [19] (en inglés, *Neural Networks - NN*), son sistemas computacionales inspirados en una versión simplificada de las neuronas biológicas. Estas neuronas artificiales, también conocidas como **perceptrones**, están definidas por una serie de elementos: entradas, **pesos**, **bias**, **función de activación** y salida. El valor que toma esta salida, siguiendo la notación de la Figura 2.1, viene dado por la ecuación $y = f(Wx + b)$.

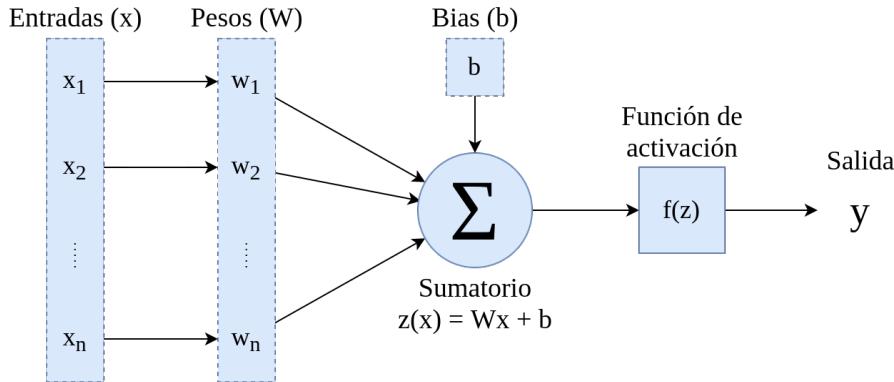


Figura 2.1: Estructura de una neurona artificial.

Una de las bases teóricas más importantes de las NN, es que con un conjunto de neuronas lo suficientemente grande y funciones de activación no lineales, es posible aproximar cualquier función continua (*Universal approximation theorem* [20]). Por desgracia, ajustar los parámetros (pesos y bias) de cada una de las neuronas de dicha red para aproximar la función deseada no es un problema trivial.

Una de las soluciones más comunes a este problema, es, de forma iterativa, modificar los parámetros de forma que minimicen la distancia entre la salida de la red y la función que se busca aproximar. Para esto, se emplea la propagación hacia atrás (*backpropagation*) [21], que necesita: un **optimizador** (en los que se profundiza más adelante), un **conjunto de entradas con salidas conocidas** y una **función de pérdida** que se emplea como aproximación optimizable de la distancia previamente mencionada. Este algoritmo, de forma simplificada, consiste en:

1. Calcular la salida de la red a partir de las entradas de las cuales conocemos la salida esperada (**forward pass**).
2. Emplear una **función de pérdida** para calcular una medida del error entre la salida obtenida y la salida esperada.
3. Calcular las derivadas parciales del error respecto de cada uno de los parámetros (**pesos** y **bias**) que se quieren ajustar.
4. Ajustar los valores de los parámetros en función de su influencia en el error empleando un optimizador concreto.

Estos pasos, normalmente se repiten varias veces para cada ejemplo disponible (entendiendo como ejemplo las parejas entrada-salida conocidas). En el contexto de las redes neuronales, este proceso de ajuste se conoce como **entrenamiento** y cada una de las iteraciones que la red realiza sobre el conjunto de ejemplos se conoce como **época** (*epoch*, en inglés).

El proceso de entrenamiento, sin embargo, hay que controlarlo y validarla de alguna forma, ya que existe el riesgo de sobreajustar los parámetros de la red neuronal a los datos de entrenamiento, siendo el modelo incapaz de generalizar a datos no antes vistos. Este sobreajuste se conoce en inglés como **overfitting**. Para controlar si el modelo se está sobreajustando, es común dividir el conjunto de datos en tres subconjuntos: entrenamiento, validación, y evaluación. El conjunto de entrenamiento, se emplea para ajustar los parámetros del modelo; el de evaluación, para comprobar si hay *overfitting* y para elegir entre distintas configuraciones de un mismo modelo o distintos modelos; y el de evaluación, para calcular las métricas finales del modelo elegido para un problema concreto. En el trabajo de Sebastian Raschka [22] se presentan distintas formas de

hacer estas particiones, en este trabajo, no obstante, se emplearán particiones ya establecidas en la literatura existente para los conjuntos de datos empleados ([Apartado 3.3.3.2](#)).

A continuación, se profundiza en algunos de los conceptos mencionados prestando especial atención a los elementos que aparecen en este trabajo.

2.2.1. Funciones de activación

Tal y como se ha visto en la [Figura 2.1](#), la función de activación transforma la suma del *bias* y el producto de los pesos y las entradas para obtener el valor de salida. En la [Figura 2.2](#), es posible observar graficadas varias de estas funciones junto con sus ecuaciones.

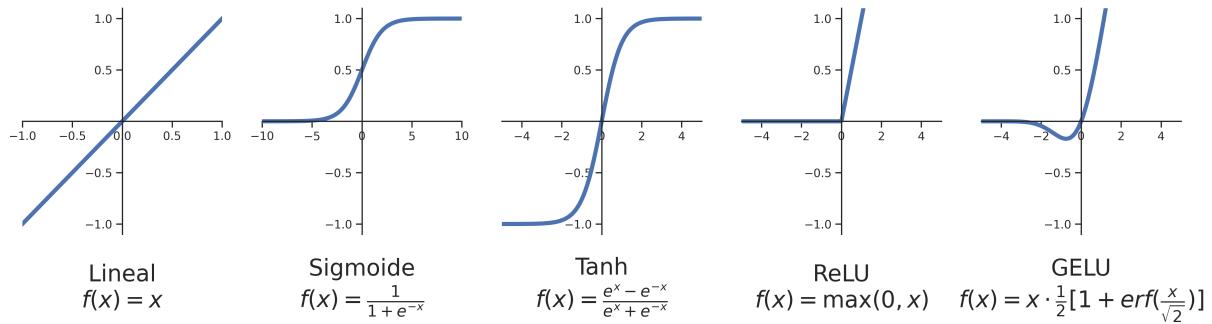


Figura 2.2: Funciones de activación.

La primera de estas funciones de activación es la función de activación lineal. Esta función no suele emplearse, ya que al concatenar múltiples neuronas con activaciones lineales, se obtiene otra función lineal que podría representarse con una sola neurona. Por esta razón, son funciones no lineales como la sigmoidal, la tangente hiperbólica, ReLU (*Rectified Linear Unit*) [\[23\]](#) o GELU (*Gaussian Error Linear Unit*) [\[24\]](#) las que suelen aparecer en las arquitecturas actuales.

Cada una de estas funciones de activación tiene características propias que las hacen diferentes entre sí. La tangente hiperbólica, aún siendo similar a la sigmoidal, es simétrica y tiene unos gradientes más fuertes en las zonas cercanas al cero, lo que origina que sus salidas sean valores pequeños centrados en cero y que el entrenamiento sea más rápido [\[25\]](#). La función ReLU, por otro lado, pese a ser parcialmente lineal, puede aproximar funciones no lineales gracias a su definición a trozos, además, el gradiente para los valores positivos es constante (1), reduciendo en gran medida el desvanecimiento de los gradientes cuando se definen redes neuronales con un gran número de capas. Por último, la función GELU, es comúnmente empleada en las arquitecturas basadas en *transformers* y mantiene el gradiente constante en valores positivos a la vez que añade una ligera ponderación a los valores negativos cercanos al cero.

2.2.2. Funciones de pérdida

Las funciones de pérdida, tal y como se ha mencionado previamente, son aproximaciones optimizables de la distancia o error entre el valor real a predecir y la predicción realizada. Dentro de estas funciones, cuyo valor se busca minimizar, existen diferentes situaciones que pueden dificultar el aprendizaje de los modelos, por ejemplo, los mínimos locales, donde los valores de la función de pérdida son menores que los valores de su entorno, originando que los gradientes se desvanezcan y detengan el proceso de optimización, o los puntos de silla (*saddle points*), donde, pese a no haber un mínimo local, los gradientes también se desvanezcan, deteniendo la minimización.

2.2.3. Optimizadores

Los optimizadores son algoritmos que minimizan el valor de la función de pérdida ($f(\theta)$) ajustando los parámetros (θ) de los modelos, entre otras cosas, en base a la influencia de estos parámetros en el error obtenido en dicha función. Esta influencia se obtiene calculando las derivadas parciales del error respecto de cada parámetro (gradiente: $\nabla_{\theta} f_t(\theta_{t-1})$), y, en función del algoritmo que se elija, puede verse afectada por distintos factores. Algunos de los optimizadores más empleados son SGD, AdaGrad [26], RMSProp [27], Adam [28] o AdamW [29].

El descenso de gradiente tradicional promedia los gradientes del error en todos los ejemplos disponibles para actualizar los parámetros. La modificación que introduce el **SGD (Stochastic Gradient Descent)**, es actualizar los parámetros tras ver cada uno de los ejemplos. De esta forma, se acelera el entrenamiento, ya que aunque cada ajuste no es óptimo, los parámetros se actualizan mucho más frecuentemente. Además, el ruido intrínseco de cada ejemplo tiene un efecto regularizador que puede ayudar a salir de mínimos locales.

No obstante, gracias a la capacidad de procesamiento paralelo de las tarjetas gráficas es muy común promediar el cálculo del error en pequeños subconjuntos (**batch**) de todos los datos disponibles. Esto se conoce como *Mini Batch Gradient Descent* y el tamaño (**batch size**) que se elija influye directamente en diferentes factores como son la velocidad de entrenamiento, la rapidez para converger, o el punto al que se converge [30].

Estos tres algoritmos, y en general la mayoría de los optimizadores, controlan el cambio de los parámetros mediante un hiperparámetro conocido como tasa de aprendizaje (*learning rate*: γ), que se multiplica por el valor del gradiente de la función de pérdida respecto de los parámetros ([Ecuación \(2.1\)](#)).

$$\theta_t = \theta_{t-1} - \gamma(\nabla_{\theta} f_t(\theta_{t-1})) \quad (2.1)$$

En el caso del *Mini Batch Gradient Descent*, al promediar los gradientes de varios ejemplos se reduce el ruido del SGD. Inspirado por el concepto físico de **Momento**, es posible añadir a estos optimizadores una modificación para que se consideren también los gradientes de los pasos anteriores del optimizador. De esta forma, se reduce el ruido y se hace más robusta la optimización frente a mínimos locales. El momento se controla con un hiperparámetro μ siguiendo la [Ecuación \(2.2\)](#), donde v_t es el Momento en un paso concreto y $g_{t,t-1}$ los gradientes correspondientes a este paso.

$$v_t = \mu v_{t-1} + g_{t,t-1} \quad (2.2)$$

Este cálculo multiplica los momentos anteriores de forma recursiva por un valor menor que uno, de forma que influyen todos los gradientes anteriores, pero cada vez con menos fuerza. Esto se puede ver comprobar matemáticamente desarrollando la [Ecuación \(2.2\)](#) ([Ecuación \(2.3\)](#)). Finalmente, este momento toma la posición del gradiente en la actualización de los parámetros ([Ecuación \(2.4\)](#)).

$$v_t = \mu^2 v_{t-2} + \mu g_{t-1,t-2} + g_{t,t-1} = \sum_{i=0}^{t-1} \mu^i g_{t-i,t-i-1} \quad (2.3)$$

$$\theta_t = \theta_{t-1} - \gamma(v_t) \quad (2.4)$$

AdaGrad (Adaptative Gradients) es otro optimizador que también modifica el descenso del gradiente. En este caso, para reducir las tasas de aprendizaje en aquellos parámetros que han tenido mayores derivadas parciales, dividiéndolas entre la suma de los gradientes (g) anteriores

(Ecuación (2.5) y Ecuación (2.6)). Estos gradientes se acumulan durante toda la optimización, por lo que es especialmente efectivo en datos dispersos donde no en todos los ejemplos se dispone de todas las características, ya que las características que no se hayan visto tan frecuentemente, tendrán actualizaciones mayores. Esto, sin embargo, también ralentiza el algoritmo, ya que según avanza el proceso, las actualizaciones de los parámetros son menores y pueden llegar a detener el proceso, impidiendo que se alcance a un mínimo aceptable.

$$\text{acumulación}_t = \text{acumulación}_{t-1} + g_t^2 \quad (2.5)$$

$$\theta_t = \theta_{t-1} - \gamma \frac{g_t}{\sqrt{\text{acumulación}_t}} \quad (2.6)$$

RMSProp (Root Mean Square Propagation) soluciona el problema de la velocidad de AdaGrad añadiendo un factor de decaimiento. Este factor de decaimiento (*decay rate*: α), escala de forma recursiva la suma acumulada de los gradientes anteriores (media móvil exponencialmente ponderada - *Exponential weighted moving averages*), por lo que a medida que el optimizador avanza, la influencia de los gradientes anteriores disminuye. De esta forma, se mantienen los beneficios de AdaGrad evitando reducir indefinidamente la magnitud de la actualización de los parámetros (Ecuación (2.7) y Ecuación (2.8)).

$$\text{acumulación}_t = (\alpha)\text{acumulación}_{t-1} + (1 - \alpha)g_t^2 \quad (2.7)$$

$$\theta_t = \theta_{t-1} - \gamma \frac{g_t}{\sqrt{\text{acumulación}_t}} \quad (2.8)$$

Por último, **Adam** y **AdamW** son dos optimizadores que fusionan las ideas y ventajas detrás del momento y de RMSProp. Para conseguir esto, se vuelve a aplicar la media móvil exponencialmente ponderada, esta vez tanto para el primer como el segundo momento, controlado por dos parámetros β (Ecuación (2.9) y Ecuación (2.10)).

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)g_t \quad (2.9)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)g_t^2 \quad (2.10)$$

Además, para reducir la influencia del valor inicial (igual a cero) de m_0 y v_0 , se añade una normalización que sigue la Ecuación (2.11) y la Ecuación (2.12).

$$\hat{m}_t = \frac{m_t}{(1 - \beta_1^t)} \quad (2.11)$$

$$\hat{v}_t = \frac{v_t}{(1 - \beta_2^t)} \quad (2.12)$$

Una vez calculados estos dos términos, se actualizan los parámetros de forma similar a como se hacía en RMSProp, con la diferencia de que esta vez se multiplica por el momento acumulado y no por los gradientes (Ecuación (2.13)).

$$\theta_t = \theta_{t-1} - \gamma \frac{\hat{m}_t}{\sqrt{\hat{v}_t}} \quad (2.13)$$

En cuanto a la diferencia entre Adam y AdamW, esta reside en cómo se implementa el *weight decay*, un tipo de regularización que penaliza los pesos mayores ya que pueden introducir mayor varianza en el modelo (*overfitting*). Tal y como se indica en el artículo *Decoupled Weight Decay Regularization* [29], la implementación del *weight decay* en los optimizadores (Adam entre ellos)

aplica este decaimiento en los gradientes. Sin embargo, al añadir las operaciones relacionadas con los gradientes adaptativos, aplicar el decaimiento en los gradientes deja de ser equivalente a aplicarlo realmente en los pesos. AdamW soluciona este problema implementando el método de regularización correctamente.

2.3. MLP

El Perceptrón Multicapa [31], o por sus siglas en inglés, **MLP** (*Multi Layer Perceptron*), es una de las arquitecturas de red neuronal más básicas y está formado por una capa de entrada, una capa de salida, y un bloque de capas intermedias u ocultas. Estas capas intermedias, son agrupaciones de neuronas donde cada una de las neuronas (normalmente) se conecta con todas las neuronas de la capa anterior y con todas las neuronas de la capa posterior (Figura 2.3). De esta forma, se crea una sucesión donde las entradas de una neurona son las salidas de las neuronas de la capa anterior y la salida de dicha neurona es una de las entradas de todas las neuronas de la capa siguiente.

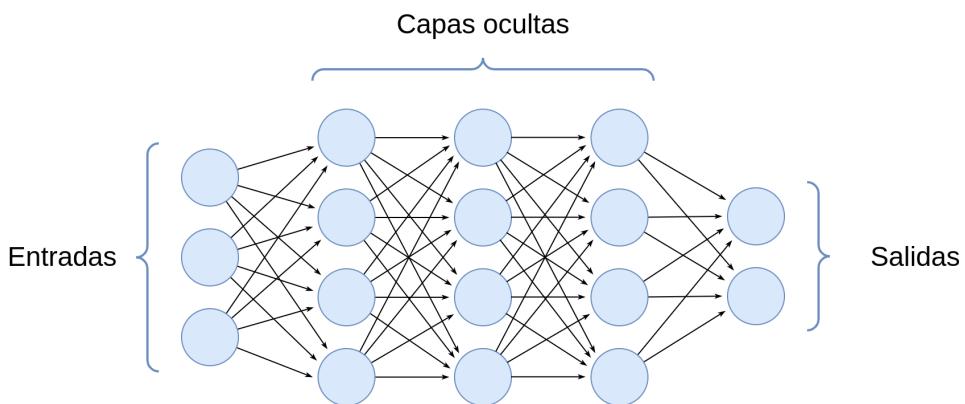


Figura 2.3: Perceptrón Multicapa.

2.4. RNN

Las redes neuronales recurrentes (*Recurrent Neural Networks - RNN*) son un grupo de redes neuronales cuya arquitectura se fundamenta en la utilización repetida de la salida de la propia red en un instante como entrada adicional de la red en el instante siguiente. Para conseguir esto, se mantiene la información en forma de estado oculto (*hidden state*). Gracias a esta transmisión recurrente, es posible transferir información entre entradas sucesivas de la red.

Este tipo de redes está especialmente diseñado para trabajar con entradas secuenciales de valores (series temporales, texto, señales de audio y video, etc.) ya que su arquitectura aprovecha la información de los datos previos y no tiene limitaciones en el tamaño de la secuencia de entrada.

Sin embargo, la ejecución y entrenamiento de este tipo de redes requiere ejecutarlas individualmente con cada uno de los valores de la entrada, haciendo que el proceso sea temporalmente costoso. Además, pese a que se han presentado distintas modificaciones como las celdas GRU (*Gated Recurrent Units*) [32] o LSTM (*Long Short-Term Memory*) [33] que los mitigan, también sufren tanto de desvanecimientos de gradientes por las activaciones sucesivas como de pérdida de información en cadenas de gran tamaño (al codificar toda la información anterior en un estado oculto, llega un punto en el que la información almacenada en instantes alejados temporalmente deja de ser representativa).

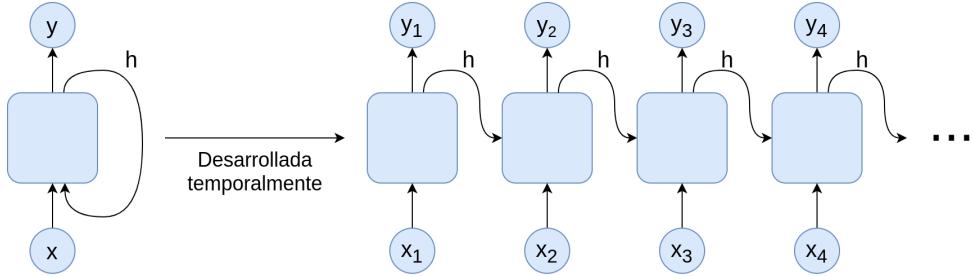
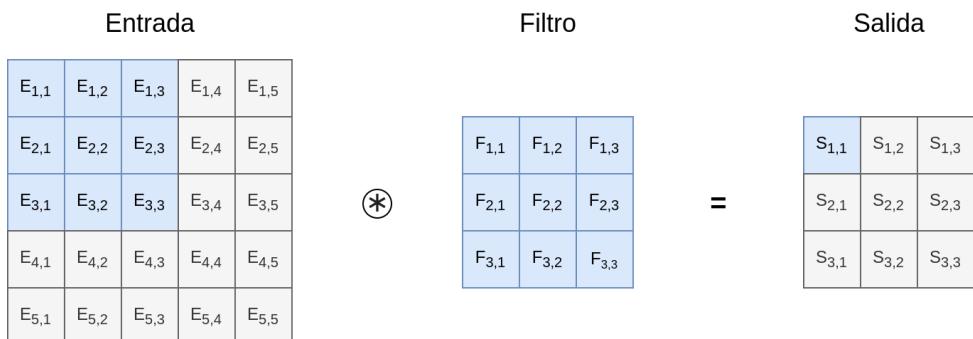


Figura 2.4: Red recurrente.

2.5. CNN

A la hora de trabajar con imágenes (matrices de dos o tres dimensiones), el número de conexiones, y por lo tanto, parámetros, que tendrían que existir para conectar cada valor de entrada con cada neurona crece enormemente. Las redes convolucionales, no solo lidian con este problema, sino que además lo hacen proporcionando muy buenos resultados. Este tipo de redes se basan en el concepto de núcleo (*kernel*) proveniente del procesamiento digital de imágenes y la operación de convolución, y su origen se atribuye al trabajo de LeCun et al. [34].

Los filtros o *kernels* (Figura 2.5) son pequeñas matrices (cuyos valores son equivalentes a los pesos vistos anteriormente, y por lo tanto son ajustados durante el entrenamiento) que se convolucionan sobre la imagen de entrada, extrayendo de esta forma mapas de características (también conocidos como mapas de activaciones) que son a su vez la entrada de las siguientes capas convolucionales tras aplicar una función de activación y puede que otras operaciones. En cuanto a la convolución (o correlación cruzada²), es una operación cuya salida se obtiene desplazando una de las dos matrices y calculando su producto escalar con la otra matriz (Figura 2.5). Estas capas, están definidas principalmente por el tamaño del *kernel*, el número de *kernels* que se aplican y la zancada o *stride*, que es el desplazamiento del *kernel* para calcular cada valor de salida y por lo tanto afecta directamente en su tamaño.

Figura 2.5: Operación de convolución con un *kernel* de 3×3 y un *stride* de 1 para uno de los valores de salida.

Los *kernels*, normalmente tienden a extraer características de más alto nivel cuanto más adelante están en la red. Esto significa que los mapas de características resultantes en las capas iniciales son características de muy bajo nivel (líneas verticales, horizontales, curvas, esquinas, etc.).

²La operación matemática de convolución entre dos funciones incluye voltear una de las funciones. Sin embargo, en el campo del Deep Learning, voltear la entrada o el filtro requeriría una operación adicional que solo conllevaría que los pesos obtenidos tras el descenso de gradiente estuvieran también volteados. Por lo tanto, la operación de convolución en redes neuronales realmente es una operación de correlación cruzada donde no se voltean ninguna de las funciones.

mientras que las de las últimas capas extraen características más complejas propias de cada dominio de aplicación [35].

En las redes convolucionales, es común encontrar otras operaciones como son:

- Pooling: Las capas de *Pooling* se emplean principalmente para reducir el tamaño de los mapas de características resultantes de las capas convolucionales. Para conseguirlo, agrupan un conjunto de valores en un solo valor aplicando una función que normalmente es el máximo (*MaxPooling* - se escoge el máximo valor) o la media (*AveragePooling* - se calcula la media de los valores). La operación de *Pooling* está definida principalmente por dos valores similares a los de la convolución: el tamaño del *kernel*, que define el tamaño del subconjunto de valores; y la zancada o *stride*, que define el número de valores que se traslada el *kernel* cada paso. En la Figura 2.6, se puede observar una operación de *Pooling* con un *kernel* de 2×2 y un *stride* de 2×2 donde cada uno de los valores de la salida se calcula a partir de los valores de la entrada con el mismo color.

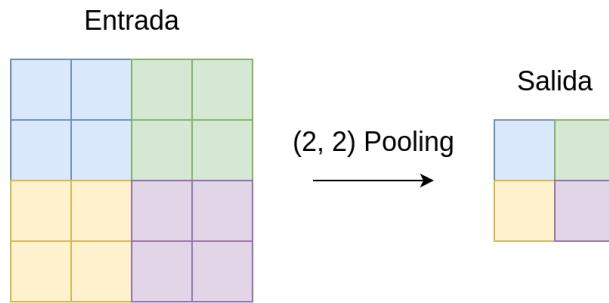


Figura 2.6: Operación de Pooling 2D.

- Convolución transpuesta: La operación de convolución puede, dependiendo del tamaño del *kernel*, producir un resultado de dimensiones espaciales (altura y anchura) iguales o menores que las de la entrada. Sin embargo, para obtener una salida con mayores dimensiones que la entrada es necesario recurrir a la convolución transpuesta, una operación similar a la convolución donde para obtener los valores de la salida, se multiplican todos los valores del *kernel* por solo uno de los valores de la entrada, que se recorren individualmente. Al hacer el cálculo de esta forma, dependiendo de la zancada (*stride*) que se elija, pueden solaparse valores de la salida, por lo que se suman aquellos que hayan obtenido un valor en más de una operación (Figura 2.7).

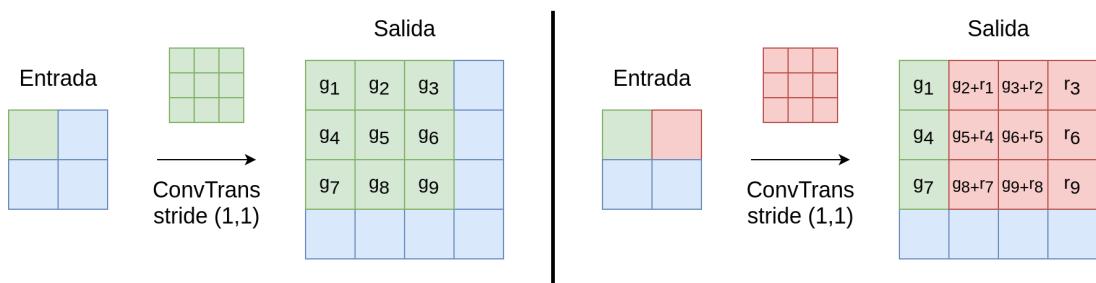


Figura 2.7: Primeros pasos de una convolución transpuesta.

- Convolución 1x1 (*Pointwise Convolution*): Pese a que no es un tipo de capa distinta per se, esta aplicación particular de la convolución suele emplearse junto a otro tipo de capas o como proyección para modificar el número de canales. Su principio de funcionamiento es el uso de *kernels* de tamaño 1×1 y tantos canales como mapas de características tenga

la entrada. De esta forma, cada uno de los filtros produce un solo canal de igual altura y anchura que la entrada en la salida, permitiendo por lo tanto controlar con el número de filtros de la capa convolucional el nuevo número de canales en el que se proyecta la entrada ([Figura 2.8](#)).

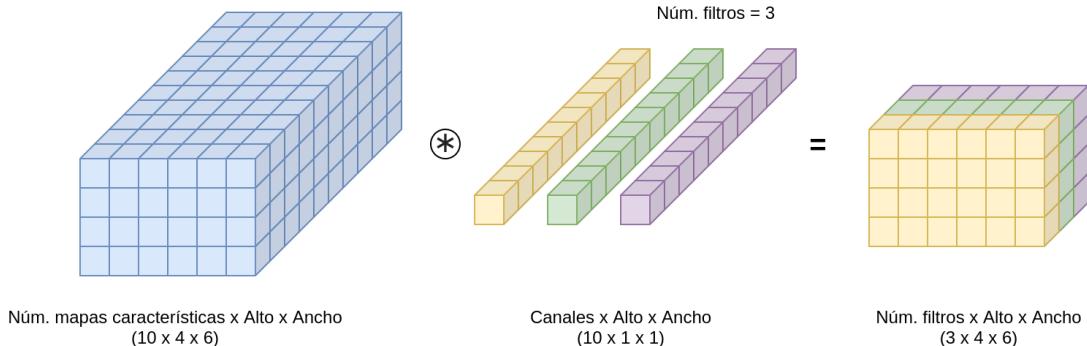


Figura 2.8: *Pointwise convolution.*

- Convolución por profundidad separable (*Depthwise Separable Convolution*): Similar al caso anterior, este cálculo es mas bien un caso especial de la capa de convolución. La operación de convolución por profundidad separable se compone de dos partes:

1. Una convolución por profundidad donde los *kernels* solamente tienen un canal y se aplican de forma individual en solo uno de los canales (el número de *kernels* tiene que ser igual al número de canales en la entrada) para luego agrupar los resultados ([Figura 2.9](#)).
2. Una convolución con un solo *kernel* de tamaño 1×1 , fusionando los canales individuales resultantes de la operación anterior en uno solo. De esta forma, la operación se asemeja a una convolución corriente en la que el *kernel* tendría tantos canales como la entrada, pero reduciendo muy considerablemente el número de operaciones.

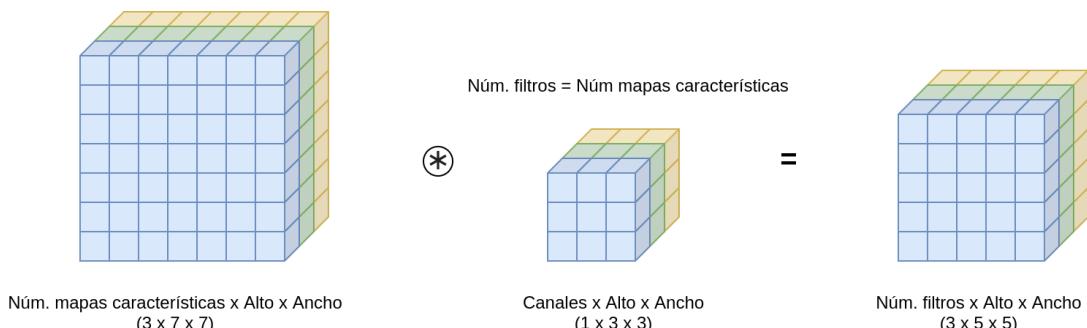


Figura 2.9: *Depthwise convolution.*

2.5.1. Conexiones residuales y ResNets

En el artículo publicado por He et al. [11] se presentan las conexiones residuales y la familia de redes ResNet, compuesta por distintos modelos en función de su número de capas. En el momento en el que se publicó dicho estudio, gracias a las conexiones residuales, se consiguió multiplicar por aproximadamente 7 el número de capas en los modelos del estado del arte (de las 22 capas de GoogleLeNet [36] a las 152 capas de ResNet152). Para conseguir esto, las ResNet se apoyan en las conexiones residuales ([Figura 2.10](#)), que permiten aumentar tanto la capacidad de aprendizaje de las redes como la calidad de sus resultados.

Las conexiones residuales están motivadas por la idea de que dada una red con una profundidad óptima para una tarea, en teoría, por muchas capas que se incluyesen, el descenso del gradiente sería capaz de hacer que las capas adicionales convergiesen en una función identidad y no perjudicasen los resultados. No obstante, empíricamente se ha comprobado que con las redes neuronales llega un punto en el que añadir más profundidad hace que los resultados empeoren. Aquí es donde entran las conexiones residuales, ya que al pasar hacia adelante la entrada de un bloque de capas y sumarlo a su salida ([Figura 2.10](#)), se facilita que el descenso del gradiente reste importancia a la señal que ha atravesado dicho bloque ($F(x)$) en caso de que no mejore la salida final de la red.

Además, para reducir el uso de recursos computacionales al incrementar el número de capas, las ResNet utilizan conexiones residuales con cuellos de botella (*bottleneck*) para reducir el tiempo de ejecución. Este sistema, en vez de usar como $F(x)$ ([Figura 2.10](#)) dos capas convolucionales, emplea tres, siendo la primera y la tercera convoluciones de filtros 1×1 que se encargan de reducir el número de canales y posteriormente incrementarlo para que se pueda sumar con la entrada. De esta forma, se reducen en gran medida el número de operaciones a realizar en la capa convolucional central, con filtros de tamaño 3×3 .

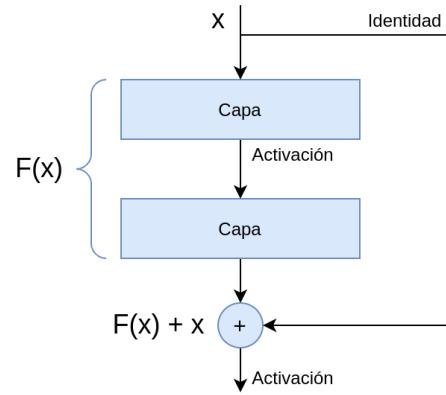
2.5.2. EfficientNet

Otro conjunto de arquitecturas, es el presentado en el artículo *EfficientNet: Rethinking Model Scaling for Convolutional Neuronal Networks* [12], que propone un método para escalar de forma proporcional la profundidad (número de capas), anchura (número de filtros) y resolución de entrada de las redes convolucionales de forma que se puedan adaptar los modelos a los recursos disponibles. En esta publicación, se presenta la red **EfficientNet-B0** (initialmente diseñada para ilustrar el escalado proporcional propuesto) y su familia de redes, que se obtienen escalando dicho modelo. Sin embargo, más allá de ilustrar el escalado en anchura, profundidad y resolución, la arquitectura de la familia de EfficientNets obtiene resultados muy competitivos con un número de parámetros muy reducido en comparación con el de sus competidores.

Para diseñar esta red se extiende el trabajo de Tan et al. [37], y se emplea *Neural Architecture Search - NAS* (una técnica que automatiza la búsqueda y configuración de arquitecturas de redes neuronales) buscando optimizar el *accuracy* y el número de operaciones en coma flotante (FLOPS). Como resultado, se obtiene una arquitectura basada en bloques de capas MBConv (*Mobile Inverted Bottleneck Residual Convolution*) [38] y *Squeeze-and-Excitation* [39].

La primera de estas dos capas, (MBConv) se propone en la publicación de **MobileNetV2** [38] y modifica los bloques con conexiones residuales y cuello de botella invirtiéndolos. En vez de utilizar filtros de 1×1 para reducir el número de mapas de características y después aumentarlo, este bloque emplea dichos filtros para aumentar y luego reducir el número de canales. Sin embargo, emplea convoluciones por profundidad separables (*Depthwise Separable Convolutions*) para reducir enormemente el número de operaciones al aumentar el número de canales.

La segunda, es uno de los bloques principales de la **SENet** [39] y basa su funcionamiento en dos operaciones que realmente pueden añadirse a cualquier otra capa: el mecanismo de *squeeze*, que crea un descriptor de cada canal de la entrada agregando sus características espacialmente; y el mecanismo de *excitation*, que toma como entrada los descriptores anteriores y los utiliza para modular los pesos de sus canales correspondientes.



[Figura 2.10: Conexión residual básica.](#)

2.6. Transformers

La arquitectura inicial del *transformer* (Figura 2.11), propuesta en *Attention is All You Need* [1], se basa en una estructura *encoder-decoder*. Es decir, un conjunto de capas (*encoder*) que codifica la entrada en una representación latente, que después es tomada como entrada del *decoder*, otro conjunto de capas que decodifica esta representación latente en una salida útil. En la propuesta inicial, destinada a procesamiento de lenguaje natural, el *encoder* se encarga de trasladar una secuencia de entrada (x_1, \dots, x_n) que representa una frase en una secuencia de representación (z_1, \dots, z_n). Esta secuencia z , es la entrada del *decoder*, que la convierte en una secuencia de salida (y_1, \dots, y_m) que representa otra frase distinta.

Una de las principales diferencias frente a los modelos *encoder-decoder* basados en redes recurrentes, es que mientras que en las redes recurrentes la entrada tiene que procesarse de forma secuencial para poder obtener los estados ocultos de las entradas anteriores, en los *transformers* la secuencia de entrada no está alineada temporalmente con la ejecución del modelo, ya que se procesa toda la entrada simultáneamente. De esta forma, se acelera muy significativamente el entrenamiento y la inferencia del modelo.

2.6.1. Arquitectura

En el **encoder** (parte izquierda de la Figura 2.11), se encuentra un *stack* de 6 capas. Cada una de estas, está compuesta por dos subcapas: una capa de *Multi-Head Self-Attention* (un mecanismo de atención que se explicará más adelante); y una capa que contiene una red *feed-forward* totalmente conectada. Cada una de estas subcapas, cuenta además con una conexión residual, que conecta la entrada de la subcapa con su salida de forma que puedan ser sumadas y normalizadas. Para facilitar la suma y normalización de entradas y salidas, todas las capas del modelo producen elementos de dimensión $d = 512$. Antes de estas 6 capas, cada uno de los *tokens* - elementos de la secuencia - de entrada (en la propuesta inicial, palabras), se convierten a vectores de dimensión d a través de un *embedding*³ previamente entrenado y se les añade una codificación posicional (en esta propuesta, generada a partir de funciones seno y coseno de distintas frecuencias) que aporta al modelo información sobre la posición de cada *token* dentro de la secuencia inicial.

Por otro lado, en el **decoder** (parte derecha de la Figura 2.11), se vuelve a encontrar un *embedding* previamente entrenado que transforma las salidas del modelo desplazadas una posición. Al resultado de este *embedding*, se le añade una codificación posicional similar a la del *encoder*. A continuación, hay otro *stack* de 6

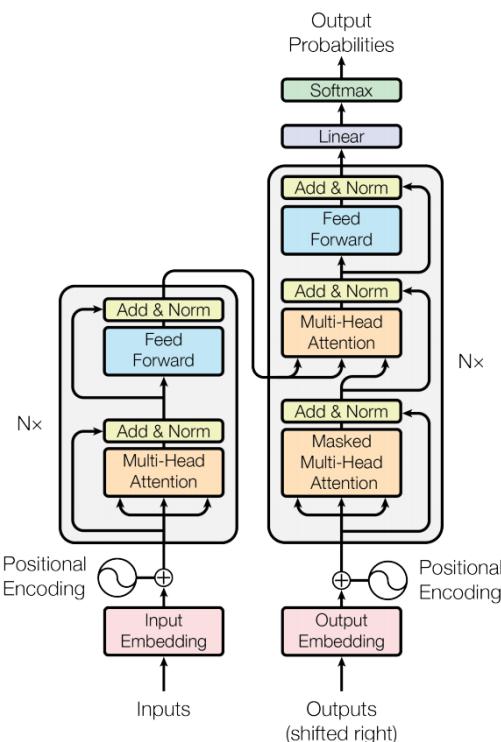


Figura 2.11: Arquitectura del *transformer*.

Fuente: [1]

³Operación que transforma, en el caso de la publicación original, palabras, en una representación numérica en un espacio vectorial donde las palabras con significado similar se encuentran próximas entre sí

capas, que esta vez está compuesto por las dos subcapas que están presentes en el *encoder* (en este caso la capa de *Multi-Head Self-Attention* es en realidad *Multi-Head Masked Self-Attention* ya que se aplica una máscara para evitar que influyan en la red los *tokens* siguientes al *token* que se va a predecir), y una subcapa adicional de *Multi-Head Cross-Attention*, situada entre las otras dos subcapas, donde las salidas de la subcapa de *masked self-attention* del *decoder* pueden acceder a las salidas del conjunto de capas del *encoder*. (**La entrada de esta última capa de atención proviene de la última capa del encoder, no de sus capas intermedias.**) Por último, a la salida del *stack* de capas del *decoder*, se encuentra una transformación lineal (entrenada) y una función softmax para predecir la salida de la red.

2.6.2. Mecanismos de atención

Los mecanismos de atención, presentados por primera vez en [40], buscan simular la atención cognitiva y han sido previamente empleados en redes recurrentes [41] y convolucionales [42, 43] para aprender qué partes de la entrada son más relevantes en la tarea a completar. Sin embargo, en [1], con los *transformers*, se propone por primera vez una arquitectura basada solamente en estos mecanismos.

Las funciones de atención más empleadas son la atención aditiva [40] y la multiplicativa, siendo esta última la empleada en los *transformers*, donde la atención se consigue a través de un producto escalar dentro de un bloque con múltiples cabezas llamado *Multi-Head Attention*. Estos bloques aparecen de dos formas distintas en los *transformers*:

- Bloques de *Self-Attention*, en el *encoder* y en el *decoder*, con todas las entradas dentro de sus respectivas capas.
- Bloques de *Cross-Attention*, en las capas del *decoder*, con entradas provenientes del final de la pila de capas del *encoder* y de la subcapa anterior del *decoder*.

Cada una de las cabezas que componen estos bloques basan su funcionamiento en multiplicar sus entradas por una serie de matrices W^V , W^K y W^Q que son aprendidas durante el entrenamiento, de donde se obtienen, respectivamente, vectores *Value* (V), *Key* (K) y *Query* (Q). Estos vectores, permiten que cada uno de los elementos de la secuencia de entrada, con el cálculo asociado a la atención (Figura 2.12) soliciten a través de su vector *Query* la información que determinen más importante de la secuencia. Esto se consigue al calcular el producto escalar de todos los vectores Q con todos los vectores K, que resultará mayor cuanto más alineados estén ambos vectores - mayor similitud entre las *Queries* (consultas) y las *Keys* (claves) -. A los resultados de estos productos escalares, se les aplica una función *SoftMax* para asegurar que sumen una unidad y finalmente se multiplican por los vectores V para obtener el resultado final de la atención. Los resultados de todas las cabezas, se concatenan en una sola matriz para aprovechar al máximo el procesamiento en paralelo y atraviesan una última proyección lineal.

Al usar solamente mecanismos de atención y cruzar toda la secuencia de entrada con sí misma, la longitud de esta deja de ser relevante, ya que cualquier elemento de la entrada puede obtener información de otro elemento, independientemente de lo alejado que esté (a diferencia de las redes recurrentes, donde la información de los elementos alejados perdía significado).

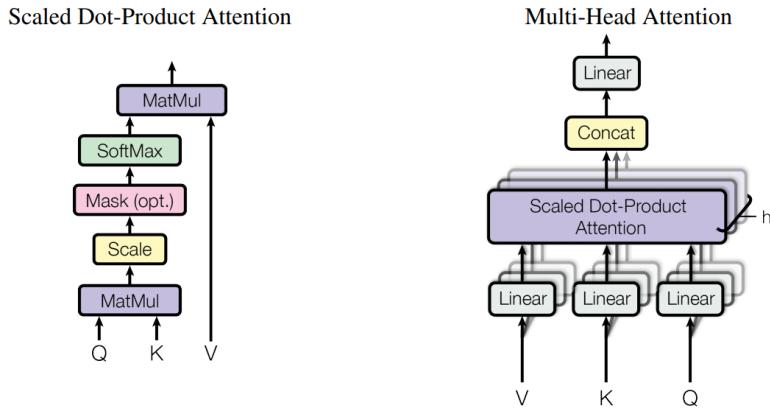


Figura 2.12: Producto escalar para el cálculo de la atención (izquierda) y bloque de *Multi-Head Attention* (derecha). Fuente: [1]

No obstante, estos mecanismos de atención son considerablemente costosos, ya que tienen una complejidad $O(n^2)$ tanto en tiempo como en memoria, siendo n el número de elementos de la secuencia de entrada (al bloque de atención). Es por esta razón por lo que han ido surgiendo una serie de propuestas para reducir dicha complejidad computacional, algunas de las cuales se exponen en la siguiente sección.

2.6.3. Atención eficiente

Para reducir los requisitos de memoria y el coste computacional de los *transformers*, que tal y como se ha mencionado anteriormente, obtienen sus resultados gracias a los mecanismos de atención, pero que sin embargo son muy costosos computacionalmente, surgen distintas técnicas que buscan aproximar el resultado de la multiplicación de matrices que se lleva a cabo en los bloques de atención. Siguiendo el esquema propuesto en [44], estas modificaciones de las capas de atención pueden agruparse en:

2.6.3.1. Patrones fijos - Fixed Patterns (FP)

En los patrones fijos, la longitud de la secuencia de entrada a los mecanismos de atención se reduce, por ejemplo: accediendo a ella en bloques de un tamaño determinado, en esto se basan ***Blockwise Attention*** [45] y ***Local Attention*** [46]; accediendo a la secuencia en intervalos previamente definidos, como en ***Sparse Transformer*** [47] o ***Longformer*** [48] donde se emplean ventanas dilatadas o con un determinado *stride* (zancada); o también empleando operaciones de *pooling* para reducir la longitud de la entrada, en ***Compressed Attention*** [49].

2.6.3.2. Combinación de patrones - Combination of Patterns (CP)

Este grupo de métodos, se basa principalmente en la combinación de diferentes patrones de acceso a los elementos que componen la secuencia de entrada y aparece en ***Sparse Transformer*** [47] donde se combinan *Local Attention* y *strided attention* asignando la mitad de las cabezas del bloque de *multi-head attention* a cada uno de los métodos. También aparece en ***Axial Transformer*** [50], donde la atención se aplica de forma independiente en cada uno de los ejes de la entrada (en este caso, el tensor de entrada debería ser multidimensional).

2.6.3.3. Aprendizaje de patrones - Learnable Patterns (LP)

Pese a ser similar a los dos casos anteriores ya que siguen basándose en diferentes formas de acceder a la secuencia de entrada para hacerla más dispersa, estos métodos son capaces de

aprender en la etapa de entrenamiento del modelo qué patrones de acceso son más adecuados. Algunas de las propuestas que emplean este tipo de patrones son **Reformer** [51], que agrupa los elementos de la secuencia de entrada (*tokens*) empleando una medida de similitud en grupos de elementos (para posteriormente aplicar el mecanismo de atención de forma independiente en cada grupo) o **Routing Transformer** [52] que emplea k-medias para agrupar los *tokens*, ambos modelos, reducen la complejidad a $O(n \log n)$. Dentro de estos modelos también destaca **ResT** [53], que está enfocado a imágenes y emplea convoluciones separables para reducir las dimensiones de las entradas al mecanismo de atención.

2.6.3.4. Disminución de rango

Este conjunto de arquitecturas, incluyen una proyección para conseguir una aproximación de la matriz resultante del cálculo de la atención, esta aproximación, pese a tener el mismo número de elementos (filas), obtiene una representación de los vectores menor (columnas), por lo que la dimensión de la matriz pasa de ser $n \times n$ a ser $n \times k$, con la consecuente disminución de coste computacional. El principal ejemplo de este tipo de arquitectura es **Linformer**, [54] que presenta una complejidad $O(n)$

2.6.4. Performer

Dentro de este trabajo, es de especial interés el mecanismo de atención eficiente propuesto por Choromanski et al. en *Rethinking Attention with Performers* [2], que reduce la complejidad de la atención de $O(n^2)$ a $O(n)$.

Para conseguir esta reducción de la complejidad, el método modifica el orden de multiplicación de las matrices Q (*Queries*), K (*Keys*) y V (*Values*) del mecanismo de atención estándar. Siendo n el número de *tokens* de dimensión d de la cadena de entrada, si en vez de multiplicar $(Q \times K^T) \times V$ se multiplica $Q \times (K^T \times V)$, las dimensiones multiplicadas serían $(n \times d) \times ((d \times n) \times (n \times d))$ en vez de $((n \times d) \times (d \times n)) \times (n \times d)$. Es decir, el número de operaciones en la multiplicación pasaría de ser $2n^2d$ (atención estándar) a ser $2d^2n$ (*performer*), haciendo que el coste crezca de manera lineal con el número de *tokens* si $n \gg d$.

No obstante, hay un problema importante, y es que antes de multiplicar la matriz de atención ($Q \times K^T$) por la matriz V , se le aplica una función *softmax* (Figura 2.12) que al ser no lineal, impide la reasociación de la multiplicación de matrices. Para solucionar esto, los autores presentan un método basado en *kernelización* y *random Fourier features* que permite aproximar el resultado de aplicar la función *softmax* a matriz de atención con dos matrices Q' y K' (de dimensiones $n \times r$) sin necesidad de calcular $Q \times K^T$ para así poder modificar el orden de las operaciones (Figura 2.13).

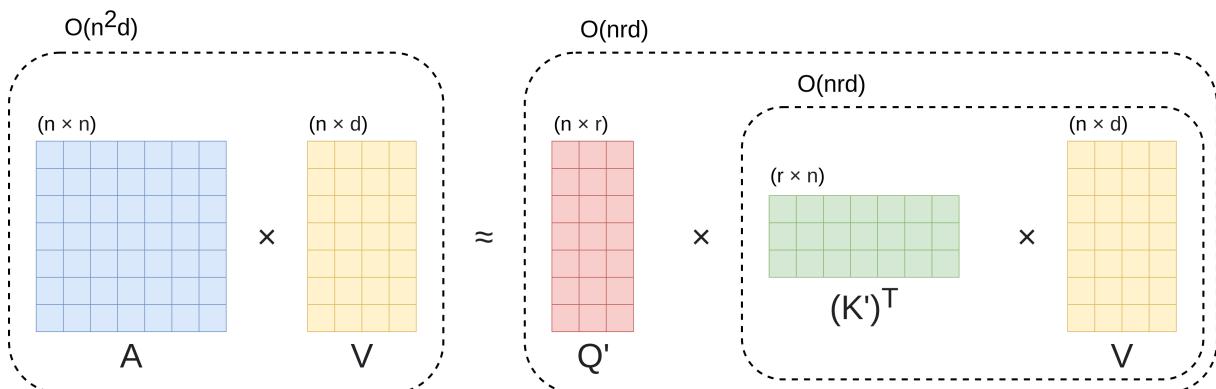


Figura 2.13: Aproximación y reasociación del orden de multiplicación en el mecanismo de atención del *Performer*. Figura inspirada en [2].

2.7. Transformers para visión artificial - ViT

A la hora de aplicar la arquitectura de los *transformers* a procesamiento de imágenes, surge un problema importante. La complejidad del mecanismo original de atención es $O(n^2)$, siendo n el número de elementos en la secuencia de entrada, por lo que para una imagen de dimensiones $lado \times lado$, el número de píxeles que conformarían la secuencia de entrada al mecanismo de atención es $n = l^2$, disparando la complejidad de la atención a $O(l^4)$. Para lidiar con este problema, se han propuesto distintas soluciones como limitar el mecanismo de atención al entorno de cada píxel (presentado en *Image Transformers* [55]) o aplicar convoluciones para reducir el tamaño de la secuencia de entrada [56].

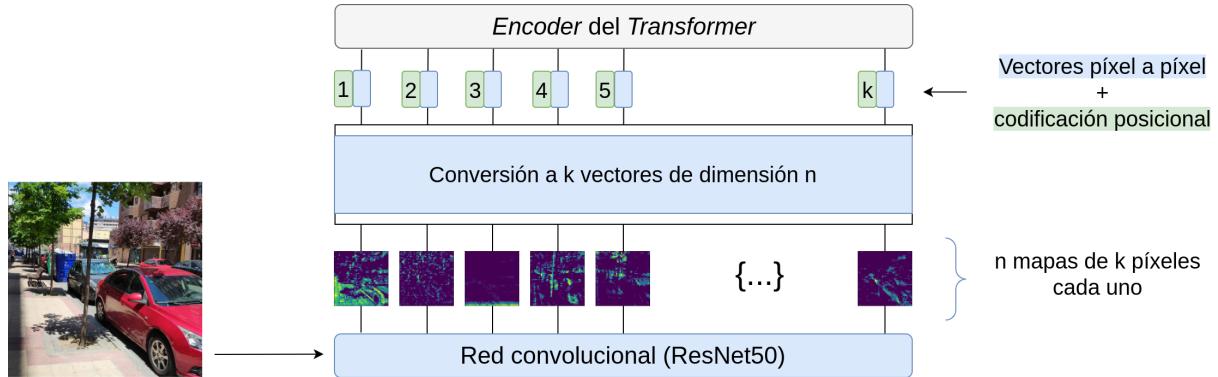
Sin embargo, la solución propuesta en *An Image is Worth 16x16 Words* con el *Vision Transformer* [3] es la que mejores resultados ha obtenido y por lo tanto aquella que más popularidad ha ganado como base de arquitecturas para otros problemas de visión artificial [4, 57, 58, 59]. Esta solución, consiste en dividir la imagen original en fragmentos de tamaño fijo y convertir con una proyección cada fragmento en un vector de valores (*embedding*). Estos vectores, son equivalentes al resultado del *embedding* de palabras en la arquitectura original y se introducen al modelo como tal, es decir, cada fragmento extraído de la imagen correspondería a una palabra de una frase (Figura 2.14). Esta arquitectura, al ser inicialmente propuesta para realizar clasificación, solamente emplea el *encoder* del *transformer*.



Figura 2.14: *Backbone* basado en fragmentos y proyección lineal del ViT.

Figura inspirada en [3]

En esta publicación, además, se presenta una variante del *Vision Transformer*, el ***Hybrid Vision Transformer***. La diferencia entre esta arquitectura y la original es la sustitución del *backbone* del ViT, es decir, de la separación en fragmentos y la proyección lineal, por una red convolucional de cuyos mapas de características se extraen los vectores que pasan a los bloques de atención (Figura 2.15). Para formar estos vectores, en la implementación usada en este trabajo, se agrupan los mapas de características en función de sus píxeles, por lo tanto, si hay n mapas de características, el píxel número 1 de cada uno de ellos formará un vector de dimensión n . De esta forma, la dimensión del vector viene dada por el número de mapas de características y el número de *tokens* será equivalente al número de píxeles de los mapas de características (imágenes mayores conllevan un mayor número de *tokens*).

Figura 2.15: *Backbone* convolucional del ViT-Hybrid.

2.8. Estimación de profundidades

Otra de las bases fundamentales de este trabajo es la estimación de profundidades. Esta tarea, tanto cuando es llevada a cabo por humanos como por máquinas, consiste en detectar la distancia relativa entre todo aquello que se ve. Tal y como se ha mencionado en el capítulo de Introducción, el cerebro humano se apoya principalmente en la disparidad existente entre las imágenes que capturan cada uno de los ojos (estereovisión), ya que las cosas que están más lejanas ven su posición menos alterada entre la vista de un ojo y de otro que las cosas cercanas. Sin embargo, también capaz de estimar la profundidad en una sola imagen a partir de nuestro conocimiento previo del mundo y del entorno (pistas monoculares).

Las técnicas de visión artificial tradicional (basadas en dos cámaras - visión estereoscópica) no pueden lidiar con este problema, por lo que la estimación monocular (con una sola cámara) es prácticamente imposible empleando técnicas tradicionales. Para este tipo de estimación (monocular), entran en juego los modelos de aprendizaje automático, que han demostrado una gran capacidad para explotar el conocimiento sobre el entorno en todo tipo de tareas, no siendo la estimación de profundidades diferente.

A continuación, aunque haciendo especial hincapié en aquellos basados en aprendizaje automático, se enumeran los distintos enfoques existentes, incluyendo aquellos propios de la visión artificial tradicional, las soluciones *hardware*, y las técnicas de aprendizaje profundo.

2.8.1. Técnicas de estimación de profundidades

La estimación de profundidades se ha intentado resolver de múltiples maneras [60]. Dentro de estas metodologías, existen tres enfoques principales en función del tipo de *software* o *hardware* que se emplea.

- **Soluciones geométricas:** Este grupo de métodos extrae información de las restricciones geométricas que existen entre parejas de imágenes. Principalmente se agrupan en técnicas de *SfM* (*Structure from Motion*) donde se reconstruye la tercera dimensión a partir de imágenes tomadas por una sola cámara en movimiento, y en técnicas de estereovisión, donde la profundidad se obtiene de la disparidad entre las imágenes capturadas por una pareja de cámaras con posiciones relativas conocidas. Estos métodos, sin embargo, basan su funcionamiento en el emparejamiento de puntos claves (*feature points*) que deben encontrarse en ambas imágenes, requiriendo por lo tanto texturas o formas características que poder emparejar.
- **Soluciones hardware:** Por otro lado, existen una serie de soluciones basadas en distintos sensores como son los LIDAR, las cámaras de tiempo de vuelo (ToF) o los escáneres de luz

estructurada. Estas soluciones, no obstante, cuentan con ciertas limitaciones como son la densidad de sus capturas (representaciones de puntos dispersos en el caso del LIDAR⁴), la sensibilidad a la iluminación (en las cámaras ToF) o su rango de acción y la necesidad de un entorno controlado (luz estructurada).

- **Aprendizaje automático:** Debido a las restricciones de los métodos existentes y a los resultados obtenidos empleando aprendizaje automático en otros campos de la visión artificial, en los últimos años han surgido múltiples arquitecturas enfocadas a recuperar la profundidad a partir de imágenes. Este documento, pese a revisar superficialmente otras opciones, se centra en las **soluciones monoculares** debido al interés detrás de obtener una representación de la profundidad a partir de una sola cámara (coste, espacio, consumo, etc.).

2.8.2. Aprendizaje automático no supervisado

Estas propuestas, ofrecen soluciones que emplean datos sin etiquetar debido a la dificultad que entraña la obtención de este tipo de anotaciones. Normalmente, estas soluciones emplean secuencias (vídeos) de imágenes monoculares, extrayendo automáticamente a partir de estas una señal supervisora con distintos métodos. En [61] se propone una arquitectura que emplea una red neuronal para calcular la posición (y movimiento) de la cámara entre *frames* consecutivos (*ego-motion*), que a su vez se emplea para calcular la profundidad de la imagen. Sin embargo, da por hecho que ningún objeto ha cambiado de posición y que el entorno es estático.

Esto, no es aplicable a entornos reales, por lo que surgen diferentes arquitecturas que generan máscaras, tanto basadas en redes neuronales [61, 62] como en técnicas geométricas [63, 64], para lidiar con los objetos dinámicos. Otros enfoques, con la misma idea subyacente, sustituyen la red de estimación de posición con métodos de odometría visual⁵ tradicional [65], que puede aportar posiciones más exactas y mejorar el funcionamiento final. Por último, algunos enfoques calculan elementos adicionales como por ejemplo el flujo óptico de la escena (cálculo de los patrones de movimiento para cada punto de la imagen) que aporta información relevante sobre la posición relativa de los objetos [62, 66].

2.8.3. Aprendizaje automático semisupervisado

Debido también a la escasez de datos etiquetados, también son populares los enfoques semi supervisados. Estas soluciones, emplean información parcial como señal supervisora, que complementan con información no etiquetada. Algunos ejemplos característicos de este tipo de aprendizaje son:

- Síntesis artificial de parejas a partir de imágenes monoculares para emplear técnicas de estereovisión, entrenando los modelos con parejas de imágenes obtenidas por conjuntos de cámaras preparados para estéreo [67, 68].
- Generación de los mapas de disparidad entre parejas de imágenes para estereovisión [69, 70], donde la señal supervisora es la disparidad obtenida mediante técnicas tradicionales de estereovisión.
- Utilización de datos etiquetados de forma dispersa (frecuentemente obtenidos con dispositivos LIDAR), que o bien emplean imágenes no etiquetadas para densificar dichas

⁴Por representaciones dispersas se entiende la captura de la profundidad en forma de una nube de puntos discretos separados entre sí, mientras que una representación densa tiene una mayor cantidad de puntos de información.

⁵La odometría agrupa aquellas técnicas que estiman el cambio de posición a partir de las lecturas de sensores. En el caso de la odometría visual, estos cambios de posición se estiman empleando principalmente imágenes capturadas por una cámara.

representaciones [71, 72, 73] o añaden la información del LIDAR en la función de perdidas para después inferir a partir de imagen monocular [74].

2.8.4. Aprendizaje automático supervisado

Pese a la dificultad para obtener datos etiquetados correctamente, los enfoques supervisados son los que mejores resultados ofrecen y por lo tanto han sido y siguen siendo extensamente estudiados. El primero de estos modelos fue propuesto en [75], y empleaba dos conjuntos de capas, uno para generar una estimación tosca de la profundidad y otro para refinar esa primera estimación. Enfoques posteriores propusieron modificaciones en la función de perdidas, para fomentar la consistencia en las predicciones [76] a través del cálculo de los gradientes de la diferencia entre el resultado y el objetivo; o mecanismos adicionales para transportar la información a través de la arquitectura. Un claro ejemplo de este transporte de información es la arquitectura **BTS** [77], donde se usan capas guía para dirigir las características extraídas en etapas iniciales de la red hasta etapas más avanzadas (*Local Planar Guidance*).

Además de aquellos basados en arquitecturas *encoder-decoder*, también hay propuestas con arquitecturas de aprendizaje adversario [78] donde el discriminador trata de distinguir entre la profundidad generada y la real [13].

Todas estas soluciones, basan su funcionamiento en redes convolucionales, sin embargo, los *transformers* han presentado muy buenos resultados en los últimos años y también se han propuesto soluciones que emplean estas arquitecturas, en el momento de redactar este documento, predominan tres: los **Dense Prediction Transformers** (DPT) [4], **AdaBins** [57], y **GLPDepth** [79]. El primero de estos dos modelos, al ser la arquitectura que se modifica a lo largo del proyecto, se explica en mayor profundidad en la siguiente sección.

El segundo, **AdaBins**, está compuesto de un *encoder* convolucional y una etapa inspirada en los *Vision Transformers* [3] que recibe la salida de el *encoder* y clasifica cada píxel en un histograma de profundidades cuyas barras (*bins*) son parametrizadas (centro y rango) dinámicamente para cada imagen. El resultado final se consigue con la suma ponderada de la predicción de pertenencia a cada barra y el valor medio de dicha barra, consiguiendo así una estimación de profundidad suavizada.

El tercer modelo, **GLPDepth**, está formado por una arquitectura *encoder-decoder* donde el *encoder* es un *transformer* y el *decoder* un conjunto de capas convolucionales. La novedad que proponen Kim et al. [79] es el uso de interconexiones (*Global-Local Paths*) que conectan el *encoder* y el *decoder* de forma que el primero obtenga información global de la imagen y cada bloque del segundo pueda fusionar esa información con las características locales (ya que son convolucionales) extraídas en sus respectivas capas.

2.8.5. DPT

Tal y como se ha mencionado previamente, la gran mayoría de arquitecturas de estimación de profundidades se basan en redes convolucionales, normalmente de tipo *encoder-decoder*. Las líneas de investigación se centran en el *decoder* y sus estrategias de agregación de información. DPT (*Dense Prediction Transformer*) [4] también se estructura como un *encoder-decoder*, pero centra su estudio en la modificación del *encoder* debido a la gran influencia que tiene este en la información que llega a la segunda parte de la red, sustituyéndolo por una arquitectura basada en mecanismos de atención (*transformer*). De esta forma, gracias a las características propias de estos modelos ya comentadas, la arquitectura se beneficia de un campo receptivo que permite atender a la información de toda la imagen en las etapas de atención. Es decir, se pasa de un campo receptivo local mucho más limitado (en las redes convolucionales) a un campo receptivo global.

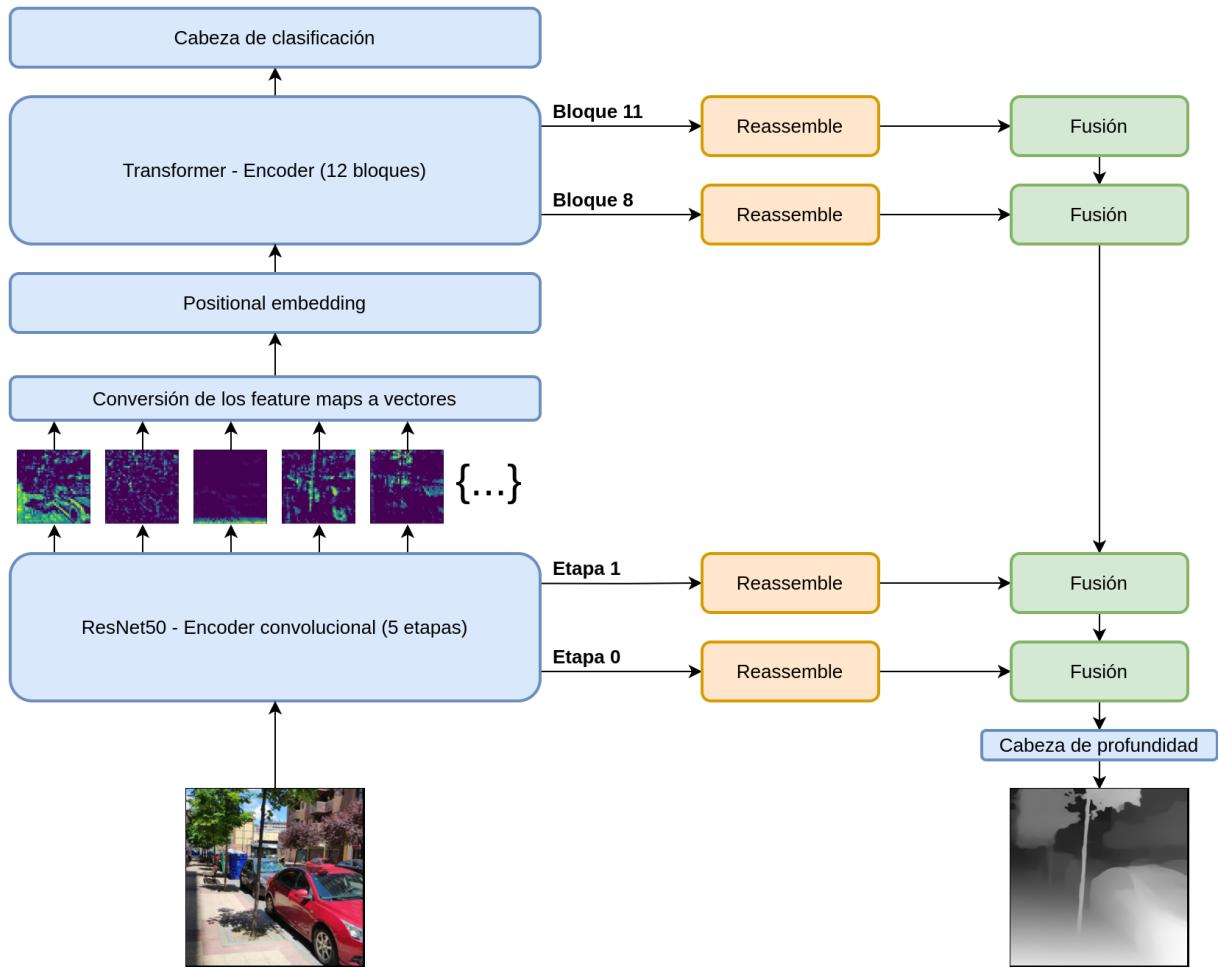


Figura 2.16: Esquema de la arquitectura de DPT-Hybrid.

En la publicación que presenta este modelo se utilizan tres variantes del *Vision Transformer* como *encoder* para llevar a cabo los experimentos: ViT-Base, con *embedding* de las imágenes a partir de fragmentos y 12 bloques de atención; ViT-Large, con el mismo método de *embedding* (esta vez con vectores de mayor dimensión para representar cada uno de los fragmentos codificados) y 24 bloques de atención, y por último, ViT-Hybrid, con el mecanismo de *embedding* de las entradas basados en una red convolucional (ResNet50) tal como se indica en la [Figura 2.16](#), seguido de 12 bloques de atención.

Por otro lado, el *decoder* de la arquitectura está compuesto por una serie de capas convolucionales que transforman representaciones extraídas de distintas etapas del *encoder*. En el caso del ViT-Base, son las salidas de los bloques de atención {3, 6, 9, 12}, en el ViT-Large, las correspondientes a las salidas de los bloques de atención {5, 12, 18, 24}, y por último, en el caso del ViT-Hybrid, son las representaciones obtenidas de los bloques de atención {9, 12} junto con las representaciones a la salida de la primera y segunda etapa de la ResNet50 encargada del *embedding*. Los puntos de dónde se extraen estas representaciones se denominan *hooks*.

Estas representaciones, están en forma de *tokens* en el caso de los bloques de atención, y en forma de mapas de características cuando se extraen de la ResNet50. Ambos tipos atraviesan un bloque *Reassemble* donde la información se concatena para reconstruir la estructura de imagen ([Figura 2.17](#)), se proyecta en el número de canales deseado, y se escala a una resolución determinada en función de la altura del *encoder* de la cual provienen (cuanto antes se extraen las características, menos se reduce su tamaño, haciendo una especie de pirámide de resolución).

Una vez se dispone de los datos en un formato similar a imágenes (a la salida de los bloques de *Reassemble*), pasan a una etapa similar a la arquitectura RefineNet [80], que duplica el tamaño de cada una de las representaciones y las suma de forma escalonada (Figura 2.17 y Figura 2.16) para fusionar su información.

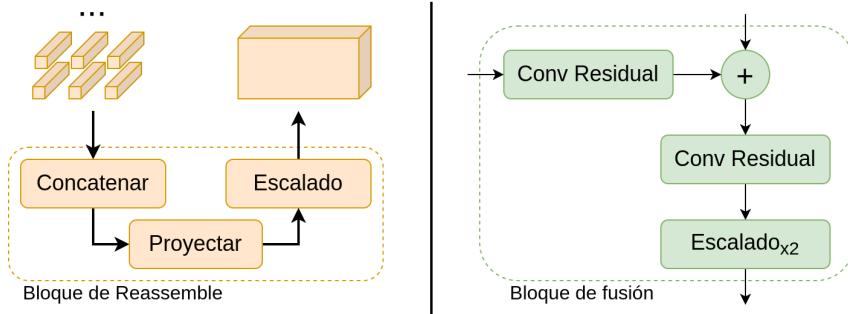


Figura 2.17: Bloques base del *decoder* de DPT. Izquierda: Bloque de *Reassemble*. Derecha: Bloque de fusión. Figura adaptada de [4].

El resultado de este último bloque (una imagen con la mitad de la resolución de la entrada), pasa a una última etapa (cabeza especializada⁶) para proporcionar la predicción final.

Antes de pasar a la metodología empleada en el trabajo original de DPT para entrenar los modelos, cabe mencionar que la red, tal y como se comenta a continuación, se entrena con una función de pérdida invariante a la escala y al desplazamiento. Esto significa que la profundidad obtenida del modelo es relativa y no un valor métrico medible. No obstante, los autores extraen valores de desviación y escala (ajustando por mínimos cuadrados las predicciones y las anotaciones en el conjunto de entrenamiento) con los que es posible transformar las profundidades relativas obtenidas en entradas nuevas a profundidades métricas (siempre y cuando la naturaleza de las entradas sea la misma que la de los ejemplos utilizados para obtener los valores de desplazamiento y escala).

Entrenamiento Los modelos proporcionados junto a la publicación, y en concreto, el que se emplea como base de este trabajo, fueron preentrenados en MIX6 (Apartado 3.3.2) y después ajustados a otros *datasets* para su evaluación. El entrenamiento en MIX6 se lleva a cabo empleando Adam como optimizador con un *learning rate* de $1e-5$ para el *encoder* y de $1e-4$ para el *decoder* y se parte de parámetros preentrenados en Imagenet (*encoder*) y inicializados aleatoriamente (*decoder*). En cuanto a la duración del entrenamiento, el modelo se entrena durante 60 épocas, cada una con 72000 pasos y un *batch size* de 16 imágenes. Además, se voltean horizontalmente las imágenes de manera aleatoria como forma de *data augmentation*. Como función de coste, se utiliza una función de pérdida [81] invariante a la escala (relativa) y al desplazamiento de la salida junto a otra función de pérdida que compara los gradientes⁷ [82] en los valores de profundidad de la salida y la anotación.

2.9. Optimización de modelos

Los modelos de aprendizaje automático del estado del arte tienden a ser cada vez más grandes. Esto generalmente conlleva un tiempo de entrenamiento y de inferencia mayor, así como unos

⁶En la publicación original, se adapta la red para segmentación semántica y para estimación de profundidades monocular. Dado que escapa del alcance de este proyecto, no se discuten en este documento las capacidades de DPT para segmentar objetos.

⁷A la hora de entrenar en otros *datasets*, desactivan esta función de pérdida si la anotación no es profundidad densa, es decir, si hay píxeles sin anotar como ocurre por ejemplo en KITTI.

requisitos de memoria y consumos de energía mayores. Para solucionar algunos de los problemas asociados, como por ejemplo la necesidad de proporcionar buenos resultados con restricciones temporales o restricciones de *hardware* (modelos embebidos, dispositivos móviles, etc.) han ido surgiendo a lo largo de los años una serie de soluciones que tratan estos aspectos, buscando siempre perjudicar lo mínimo posible los resultados de los modelos originales. A continuación, se presentan algunas de las más generales, aplicables en una gran variedad de arquitecturas.

- **Poda / Pruning:** Consiste en eliminar de los modelos aquellas conexiones o neuronas que son redundantes o menos relevantes para la red, con el objetivo principal de reducir el tamaño del modelo, ya que las redes neuronales suelen estar sobredimensionadas y son redundantes. Al reducir el tamaño del modelo, aumentar la velocidad con la que se realiza la inferencia⁸ sin sacrificar la exactitud del modelo en exceso. Siguiendo la clasificación presentada en [83], los métodos más empleados de poda se pueden agrupar en dos grandes grupos: Basados en magnitud y basados en sensibilidad.
 - **Métodos basados en magnitud:** Estas técnicas, eliminan los parámetros basándose en su valor o en la influencia que tienen en la siguiente capa. Por ejemplo, un peso con un valor muy próximo a cero apenas influirá en la capa siguiente, y por lo tanto puede eliminarse. Estas técnicas, suelen llevarse a cabo eliminando parámetros y reentrenando la red de forma iterativa, repitiendo este proceso hasta encontrar un equilibrio entre reducción de tamaño y pérdida de exactitud. Han et al. [84] presentaron, empleando este entrenamiento recursivo, resultados donde se eliminan más de un 90 % de los parámetros aumentando solamente en unas décimas el porcentaje de error en comparación con los modelos sin podar. Además de la poda de conexiones y neuronas, existen también distintas técnicas destinadas a podar con una menor granularidad, por ejemplo, mapas de características y filtros. Algunas de estas técnicas incluyen las basadas en la varianza entre canales [85] o en el número medio de ceros que tienen los mapas de características [86] entre otras.
 - **Métodos basados en sensibilidad:** Estos métodos, a diferencia de los basados en magnitud, buscan analizar el efecto de la modificación de los pesos en la función de perdida. Para ello, suelen centrarse en aproximar los cambios en la función de perdida a través de una serie de Taylor. Esta serie de Taylor, incluye una matriz hessiana, que se obtiene a partir de las segundas derivadas de la perdida respecto de los pesos. Dado que el cálculo de la matriz hessiana es computacionalmente costoso, Lecun et al. en *Optimal Brain Damage (OBD)* [87] ignora los elementos que no están situados en la diagonal de la matriz, reduciendo la complejidad del cálculo considerablemente. Posteriormente, Hassibi et al. plantearon no descartar dichos valores en *Optimal Brain Surgeon (OBS)* [88] pero sus cálculos son prohibitivos con el número de parámetros de las arquitecturas actuales. Estas series de Taylor, también se han empleado para la poda de canales y mapas de características en redes convolucionales, tanto con aproximaciones de primer orden [89] como de segundo orden [90].
- **Quantization:** La cuantificación tiene como objetivo convertir los parámetros de las redes almacenados en 32 bits en representaciones más pequeñas como son los números enteros (normalmente en 8 bits). Esta transformación, conlleva una pérdida de calidad en los resultados de los modelos, pero reduce sus requisitos de memoria y acelera la inferencia de resultados. Esta aceleración viene dada por la velocidad a la que se pueden realizar operaciones con números enteros comparado con las operaciones con números en coma

⁸Dependiendo de las técnicas utilizadas para llevar a cabo la poda, pueden aparecer limitaciones en el hardware de uso general para trabajar con matrices dispersas, pero existen soluciones tanto hardware como software para trabajar con estos datos.

flotante. Existen principalmente tres tipos de cuantificación, dinámica, estática (estas dos se aplican sobre un modelo ya entrenado) y durante el entrenamiento (*Quantization Aware Training*).

- **Cuantificación dinámica:** En este caso, no solo se convierten los pesos del modelo ya entrenado a enteros, sino que también se transforman las activaciones buscando los parámetros de dichas conversiones de forma dinámica durante la inferencia. Este tipo de cuantificación no requiere de datos pero sin embargo no es tan rápida como las otras dos al tener que realizar las transformaciones durante la inferencia.
- **Cuantificación estática:** De forma similar al caso dinámico, las activaciones de las capas se cuantifican durante la inferencia, sin embargo, en este caso una vez se ha entrenado la red se le pasan bloques adicionales de datos con los que se estiman los parámetros de estos procesos de cuantificación para acelerar la inferencia cuando se haya desplegado el modelo. De esta forma, pese a que es necesario tener datos adicionales (no hace falta que estén etiquetados), se alcanza una mayor velocidad de inferencia.
- **Quantization Aware Training:** Por último, esta opción tiene en cuenta la cuantificación durante todo el proceso de entrenamiento simulando el efecto de la cuantificación en los pesos y activaciones de forma que influyan en la función de perdidas (las operaciones durante el entrenamiento siguen haciéndose en coma flotante). Este método, debido a la consideración de la cuantificación durante el entrenamiento, resulta en una inferencia más rápida y resultados superiores a los de los métodos anteriores. Sin embargo, no siempre es aplicable al requerir el entrenamiento del modelo.
- **Weight clustering:** El *clustering* de pesos, o *weight sharing*, agrupa los pesos del modelo en un número determinado de *clusters* para asignar a cada peso el valor del centroide de su grupo correspondiente. De esta forma, se reducen los requisitos de memoria del modelo, ya que solamente es necesario almacenar los índices que apuntan al vector de centroides, que al ser números enteros se pueden representar con un número de bits mucho menor (por ejemplo, en 8 bits, reduciendo el tamaño de la matriz de pesos original (cada uno 32 bits) a un cuarto de su tamaño).
- **Mixed-precision training:** Propuesto por primera vez en [91], el entrenamiento con precisión mixta almacena los pesos, activaciones y gradientes en formato de coma flotante de media precisión (16bits - IEEE 754) en vez de simple precisión (32 bits). De esta forma, sin perder precisión, se reducen a cerca de la mitad los requisitos de memoria en el entrenamiento, que además se ve acelerado en las últimas arquitecturas de GPUs. Para conseguirlo se llevan a cabo tres estrategias: Mantener una copia maestra de los pesos almacenada en FP32, escalar el resultado de la función de perdida y acumular ciertos resultados en FP32.
 1. **Copia maestra de los pesos en FP32:** A pesar de que en el *forward* y *backward pass* del entrenamiento se usan valores FP16, se debe almacenar una copia maestra en FP32 de los pesos, que son los empleados en el optimizador para multiplicar por gradientes y *learning rates*. Si bien es cierto que esto aumenta los requisitos de memoria de los pesos en un 50 %, el impacto en el conjunto global es mucho menor ya que el consumo de memoria durante el entrenamiento está dominado por las activaciones de cada capa.
 2. **Escalado de la perdida:** Por otro lado, al calcular los gradientes de los pesos, también podrían resultar demasiado pequeños como para representarse correctamente en FP16. Esto se soluciona escalando el resultado de la función de perdidas al finalizar el *forward pass* y antes de empezar el *backpropagation* para que tengan una magnitud

mayor. De esta forma, todos los gradientes resultan escalados por la misma magnitud al derivar y basta con reescalarlos a FP32 una vez finalizado el *backpropagation*, antes de la etapa del optimizador.

3. **Precisión aritmética:** Para poder asegurar los mismos resultados que con FP32, las multiplicaciones parciales de los productos escalares y las reducciones (sumas de todos los elementos) de vectores FP16 tienen que acumularse en valores FP32, que antes de escribirse en memoria se convierten a FP16.

El Fundamento Teórico y el estudio del Estado del Arte presentados en este capítulo constituyen la base sobre la que se desarrolla el resto del Trabajo Fin de Máster, siendo DPT ([Apartado 2.8.5](#)) la arquitectura que se va a estudiar y modificar: sustituyendo su mecanismo de atención por el del Performer ([Apartado 2.6.4](#)); alterando tanto su número de cabezas de atención como los bloques de los que se extraen representaciones intermedias (*hooks*); y reemplazando su *backbone* original (ResNet, [Apartado 2.5.1](#)) por una EfficientNet de menor tamaño ([Apartado 2.5.2](#)). Además, se probarán y evaluarán las distintas técnicas de optimización de modelos presentadas ([Apartado 2.9](#)).

3: Material y Métodos

En este tercer capítulo se introducen los materiales empleados para la realización del Trabajo Fin de Máster, tanto a nivel informático (*software* y *hardware*) como a nivel de datos. Después, se detalla las metodología seguida durante el proyecto para definir los distintos modelos de que se entrenarán, y por último se describen las herramientas empleadas durante el proceso de evaluación.

3.1. Software

Lenguaje de programación y librerías

Para el desarrollo de este proyecto, se ha elegido como lenguaje de programación **Python 3.7.10** debido a su ecosistema de librerías y código abierto orientado al aprendizaje profundo. Junto con Python, se han empleado principalmente una serie de librerías y paquetes que pueden distinguirse en dos grupos:

- Uso de CPU: **Numpy 1.20** [92], para trabajar con matrices y acelerar operaciones matemáticas; **OpenCV 4.5.2** [93], para la carga y manipulación de imágenes antes de convertirlas en tensores, **fvcore 0.1.5** [94], para calcular el número de operaciones de los modelos de aprendizaje profundo, y **Matplotlib 3.4** [95] / **Seaborn 0.11** [96], para graficar resultados y otras figuras de este documento.
- Uso de GPU: **PyTorch 1.9.0** [97], para la creación, modificación, entrenamiento y evaluación de modelos de aprendizaje profundo acelerados por *hardware* (es decir, ejecutados en GPUs); **timm (PyTorch Image Models) 0.4.9** [98], desarrollado y mantenido por Ross Whigtman, que pone a disposición del usuario un gran número modelos del estado del arte preentrenados e implementados en PyTorch; el **repositorio de DPT** [4], modelo que se modifica a lo largo del proyecto; y por último, el repositorio **performer-pytorch 1.1.3** [99] de Phil Wang, que ofrece una implementación, también en PyTorch, de la arquitectura Performer y sus mecanismos de atención.

Algunas de estas librerías tienen alternativas que podrían haberse empleado perfectamente en este proyecto. La elección más importante es probablemente el uso de PyTorch frente a Tensorflow/Keras, ya que ambas librerías permiten construir y entrenar modelos de *Deep Learning* a partir de funciones y abstracciones que representan distintos tipos de capas, funciones de activación o procesos de transformación de datos, entre otras. Además, ambos paquetes de *software* ofrecen la posibilidad de ejecutar estos modelos, así como sus entrenamientos y evaluaciones en tarjetas gráficas dedicadas (GPU), reduciendo de forma drástica el tiempo necesario para completar entrenamiento e inferencia. Esta decisión, se ha tomado principalmente

por la cada vez más frecuente elección de PyTorch en proyectos de investigación debido a su mayor flexibilidad. Consecuencia directa de esto, es que gran parte de los repositorios de código relacionados con publicaciones científicas recientes (por ejemplo, DPT) usan PyTorch para crear sus modelos.

Aceleración por *hardware*

En el párrafo anterior, se ha mencionado que PyTorch acelera por *hardware* el entrenamiento y la inferencia de los modelos de aprendizaje profundo. Para esto, se apoya principalmente en **CUDA** y **cuDNN**. El primero, es una plataforma de computación paralela desarrollada por NVIDIA para sus tarjetas gráficas dedicadas que permite desarrollar código para ejecutarlo en dichos dispositivos, aprovechando así el gran número de procesadores que tienen estos componentes. El segundo, también desarrollado por NVIDIA, es una librería de primitivas aceleradas por GPU preparadas para construir redes neuronales. Pese a que en este proyecto no se ha trabajado directamente con ninguna de estas herramientas, es necesario disponer de ellas ya que PyTorch las utiliza. Las versiones empleadas son, respectivamente, CUDA 11.1 y cuDNN 8.

Gestión y seguimiento de experimentos

Dada la naturaleza del proyecto, era de esperar que el número de experimentos y de variaciones de modelos a entrenar fuese grande, por esta razón, se elige **Weights and Biases (wandb)** [100] para gestionar y monitorizar dichas pruebas, es decir, visualizar y controlar su evolución, registrando métricas y resultados para su posterior utilización. *Weight and Biases* es un servicio de seguimiento de experimentos, gratuito para uso académico y personal, que se ejecuta en la nube, dispone de una interfaz gráfica web ([Figura 3.1](#)) y permite registrar de forma sencilla variables y métricas durante las distintas ejecuciones que se lleven a cabo. Además, ofrece también un gestor de búsqueda de hiperparámetros, donde es posible configurar los valores que se quieren probar para que wandb se encargue de inicializar los scripts de entrenamiento con las configuraciones correspondientes de forma automática y coordinada en todas las máquinas en las que se ejecute su cliente. Ya que para el entrenamiento se han empleado varios equipos en paralelo, esta última característica se ha valorado muy positivamente al compararlo con otro *software* de monitorización como por ejemplo Tensorboard.



Figura 3.1: Interfaz web de *Weights and Biases*.

Entorno de desarrollo

Para gestionar la instalación y ejecución de este conjunto de *software* en un entorno controlado, limitado, y fácilmente replicable, se ha elegido **Docker** junto al **NVIDIA Container Toolkit**. Docker proporciona una capa de abstracción virtualizando a nivel del sistema operativo. Esto significa que es capaz de utilizar el kernel de Linux de la máquina anfitrión, consiguiendo de esta forma ser mucho más rápido y eficiente que una máquina virtual. Por otro lado, el NVIDIA Container Toolkit envuelve el Docker Engine y mapea las primitivas de CUDA desde el interior del contenedor hasta el driver de la GPU del sistema anfitrión. De esta forma, la máquina anfitrión solo necesita tener actualizados los drivers de la(s) tarjeta gráfica para que puedan ser empleados de manera transparente por CUDA. Para el desarrollo, se parte de una de las imágenes proporcionadas por PyTorch con la versión de PyTorch y de CUDA necesarias donde se instalan todas las librerías requeridas.

Si bien es cierto, existen otras opciones para conseguir entornos de desarrollo funcionalmente similares: Conda, por ejemplo, también gestiona las dependencias de CUDA de las librerías de aprendizaje profundo, pero puede entrar en conflicto con las librerías instaladas usando pip (el instalador de paquetes de Python) en su mismo entorno virtual, ya que no todas las librerías están disponibles en los repositorios de conda; otra opción que nos permite usar pip sin riesgo de dañar otras instalaciones en el equipo es el uso de entornos virtuales como venv, pero estos no gestionan correctamente el software y las dependencias de los paquetes relacionados con CUDA.

No obstante, Docker ofrece una ventaja más que es decisiva, la portabilidad que ofrece entre sistemas. En caso de querer ejecutar los scripts en cloud (4.1) o en dispositivos embebidos (p.e. los dispositivos Jetson de NVIDIA, que incluyen el NVIDIA Container Toolkit) sería suficiente con usar la misma imagen para tener un entorno idéntico.

Los ficheros necesarios para crear el entorno empleado en el proyecto están disponibles tanto en el repositorio del proyecto como en el [Apéndice B](#).

Otros

Por último, para la redacción de este documento se ha empleado **LaTeX** como sistema de composición de texto, **diagrams.net** para el diseño de figuras e ilustraciones, y **BibTeX** para gestionar las referencias bibliográficas. Tanto esta memoria como el desarrollo del código relacionado con el proyecto se han llevado a cabo empleado **Git** como software de control de versiones y se pueden encontrar en los repositorios <https://github.com/guillesanbri/tfm-latex/tree/v1.0.0> y <https://github.com/guillesanbri/DPT/tree/v1.0.0-tfm> respectivamente.

3.2. Hardware

Para el desarrollo de este proyecto, se ha utilizado principalmente el Equipo 1 de la [Tabla 3.1](#). No obstante, para el entrenamiento de los distintos modelos de aprendizaje profundo, se han empleado también instancias en la nube con la configuración del Equipo 2 en la [Tabla 3.1](#).

	Equipo 1 (Sobremesa)	Equipo 2 (Google Cloud)
Procesador	AMD Ryzen 7 3800x 8 núcleos @ 3.9 GHz	Intel Xeon 4 vCPU @ 2.30 GHz
GPU	NVIDIA RTX 3070 8GB 5888 CUDA cores, 184 Tensor cores Arquitectura Ampere	NVIDIA Tesla T4 16 GB 2560 CUDA cores, 320 Tensor cores Arquitectura Turing
Memoria	32 GB DDR4	15 GB

Tabla 3.1: Especificaciones de los equipos empleados durante el trabajo de fin de máster.

3.3. Datasets

Durante este trabajo, de forma directa o indirecta, se emplean ciertos conjuntos de datos. Más concretamente, ImageNet [101, 5] y MIX6 [4] han sido empleados (no durante el desarrollo de este trabajo) para preentrenar los distintos modelos usados, mientras que KITTI [102, 103, 104, 105] se elige como conjunto de datos con el que comparar y evaluar las distintas modificaciones, implementadas y por lo tanto se utiliza para entrenar los modelos. A continuación se resumen las características de estos tres *datasets*.

3.3.1. ImageNet

ImageNet [101, 5] es un *dataset* que proporciona un gran número de imágenes etiquetadas en función de la presencia o ausencia de una serie de conceptos definidos como *synsets*. Estos conceptos siguen la jerarquía propuesta por WordNet [106], donde se agrupan palabras y categorías en función de sus relaciones semánticas. Para construir ImageNet, partiendo de una fracción de la ya mencionada estructura de WordNet, se buscaron imágenes de Internet para poblar cada una de las categorías. Estas imágenes, se filtraron y posteriormente fueron manualmente etiquetadas por humanos.

Dentro del proyecto de Imagenet, existen dos conjuntos: ImageNet21K e ImageNet1K (este último, normalmente llamado ImageNet). La principal diferencia entre estos dos conjuntos es que el primero, ImageNet21K suma más de 14 millones de imágenes clasificadas en más de 21 mil clases diferentes. Por otro lado, ImageNet1K es un subconjunto de ImageNet21K compuesto por cerca de 1.2 millones de imágenes clasificadas en 1000 categorías diferentes. Además de esto, también cuenta con anotaciones de localización de objetos (*bounding boxes*) en más de medio millón de imágenes. Debido a la gran cantidad de imágenes y la variedad de elementos que abarcan, ImageNet es normalmente empleado para entrenar las arquitecturas de aprendizaje automático profundo. De esta forma, los modelos preentrenados en ImageNet pueden ajustarse de una manera mucho más rápida y efectiva a tareas e imágenes nuevas con otros *datasets*, ya que al haber sido entrenados previamente los modelos han aprendido a extraer características generales (normalmente reutilizables) de las imágenes.

Una muestra de imágenes que conforman ImageNet está disponible en la Figura 3.2.

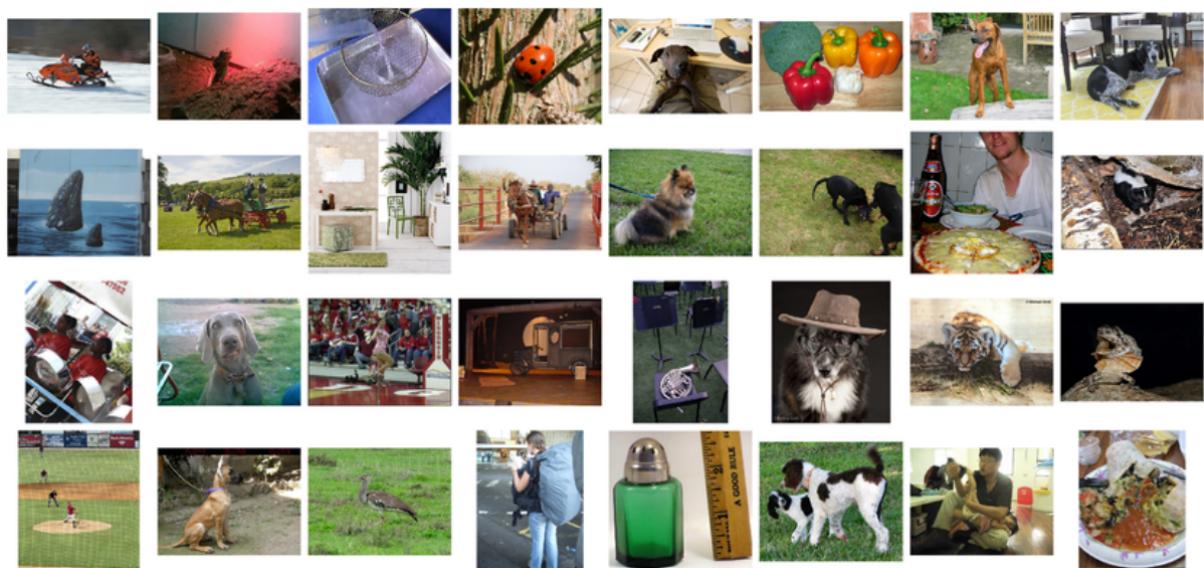


Figura 3.2: Muestra de las imágenes de ImageNet. Fuente: [5]

3.3.2. MIX6

MIX6 [4], una ampliación de MIX5 [81], es en realidad una agrupación de otros *datasets* que proporcionan anotaciones de profundidad de sus imágenes. Estas agrupaciones, consiguen dos características importantes: Primero, suman una cantidad de imágenes considerablemente alta; segundo, al tener datos de naturalezas tan distintas, existe una enorme variedad entre las imágenes, lo que permite entrenar modelos de estimación de profundidades generales, es decir, que no estén especializados en ningún entorno concreto.

Estas dos características, hacen que MIX6 sea una muy buena opción para entrenar arquitecturas basadas en *transformers* pero también dificultan el entrenamiento de modelos debido a la falta de homogeneidad entre los formatos de las imágenes, sus anotaciones, etc. Un desglose resumido de los *datasets* que componen MIX6 está disponible en la [Tabla 3.2](#).

Dataset	Descripción	Núm. de imágenes
Entrenamiento		
DIML Indoor [107]	Imágenes reales anotadas con cámara Kinect de Microsoft.	220K
MegaDepth [82]	Imágenes reales anotadas con MVS (<i>Multi View Stereo</i> - Múltiples puntos de vista en diferentes fotografías)	130K
ReDWeb [108]	Imágenes reales anotadas a partir de estereovisión.	3.6K
WSVD [109]	Vídeos recuperados de YouTube en formato de estereovisión anotados a partir de dicha pareja de imágenes.	1.5M
3D Movies [81]	Películas 3D grabadas con cámaras estereoscópicas anotadas a partir de la pareja de imágenes.	75K
TartanAir [110]	Imágenes sintéticas.	1M
HRWSI [14]	Imágenes reales anotadas a partir de estereovisión.	21K
ApolloScape [111]	Imágenes reales anotadas con sensor LiDAR.	5.1K
BlendedMVS [112]	Imágenes sintéticas.	17K
IRS [113]	Imágenes sintéticas.	103K
Evaluación		
DIW [114]	Imágenes reales anotadas manualmente con la profundidad relativa entre pares de puntos aleatorios.	495K
ETH3D [115]	Imágenes reales anotadas con sensor LiDAR.	5.2K
Sintel [116]	Imágenes sintéticas.	1K
KITTI [102]	Imágenes reales anotadas con sensor LiDAR.	45K
NYUDepthV2 [117]	Imágenes reales anotadas con cámara Kinect de Microsoft.	407K
TUM [118]	Imágenes reales anotadas con cámara Kinect de Microsoft.	87K

Tabla 3.2: Datasets que conforman MIX6. Subrayados aquellos que no forman parte de MIX5.

3.3.3. KITTI

KITTI [102, 103, 104, 105] es un proyecto desarrollado por el *Karlsruhe Institute of Technology* y el *Toyota Technological Institute* que engloba un *dataset* y un conjunto de *benchmarks* enfocados a diferentes tareas relacionadas con la conducción autónoma. Los *benchmarks* que incluye este proyecto evalúan: estereovisión, flujo óptico (*optical flow*), flujo de la escena, **estimación de profundidades monocular**, *depth completion*, odometría visual/SLAM, localización de objetos (2D, 3D y cenital), seguimiento de objetos, segmentación de carreteras, y por último, segmentación de objetos general, tanto semántica como a nivel de instancia. Debido a la naturaleza de este trabajo, este apartado se centrará en la parte referente a la predicción de profundidad monocular.

Los datos disponibles en KITTI fueron capturados empleando un vehículo equipado con diferentes sensores y realizando diferentes recorridos en distintas zonas urbanas e interurbanas. De esta forma, se capturaron escenarios variados en múltiples condiciones de luz, hora, presencia de vehículos y peatones, etc. Dentro de los sensores equipados, son de especial interés para este trabajo las dos parejas de cámaras para estereovisión (un montaje con dos cámaras en escala

de grises *PointGray Flea2 grayscale* y otro montaje con dos cámaras en color *PointGray Flea2 color*) y el escáner láser rotatorio de 360° *Velodyne HDL-64E*. Además de estos sensores, el automóvil también equipaba un sensor de medida inercial con sistema de navegación GPS para registrar información relacionada con la odometría que no se ha empleado durante el desarrollo del trabajo.

3.3.3.1. Datos

Si nos centramos en la información relevante para la estimación de profundidades monocular, KITTI ofrece los siguientes datos:

Datos en bruto

El *dataset* está compuesto por fotogramas muestreados y sincronizados a 10 Hz de los vídeos capturados por las cámaras en diferentes recorridos. Debido a las características del sistema óptico, para cada instante se disponen de cuatro imágenes, derecha e izquierda en escala de grises, y derecha e izquierda en color. Una muestra de estas imágenes puede observarse en la [Figura 3.3](#).



Figura 3.3: Muestra de las cuatro imágenes en bruto disponibles en KITTI para un instante dado.

En total, se disponen de 192760 imágenes (~ 196 GB) de tamaño 1242x375 píxeles, de las cuales 96430 (la mitad) corresponden a las cámaras a color. Como el objetivo es la estimación de profundidades monocular, solo se emplea una de las imágenes de cada pareja de imágenes producido por el sistema de estereovisión, por lo que realmente se emplean 48215 imágenes de los datos en bruto (una cuarta parte de la cantidad original).

Anotaciones

Por otro lado, KITTI proporciona también los valores numéricos de la profundidad para cada uno de los píxeles (de las imágenes presentadas previamente) almacenados como imágenes en formato PNG con un solo canal y 16 bits para cada valor (UINT16). Estos valores son los obtenidos por el escáner láser equipado en el vehículo y pueden considerarse una medida fiable de la profundidad en cada imagen. Por lo tanto serán los datos que se emplearan como anotaciones para entrenar los modelos y evaluar sus capacidades de estimación de profundidades. Un punto importante a considerar sobre las medidas de estas anotaciones es que debido a la naturaleza del sensor con el que fueron tomadas, son anotaciones **dispersas** (*sparse*), no densas. Esto significa que no todos los píxeles de una imagen tienen anotación, y por lo tanto aquellos píxeles no anotados deberán ser ignorados tanto durante el entrenamiento como durante la evaluación. Una muestra de estas etiquetas y de las anotaciones dispersas puede observarse en la [Figura 3.4](#). Estas anotaciones

están disponibles tanto como para las imágenes capturadas con las cámaras derechas como para las capturadas con las cámaras izquierdas.

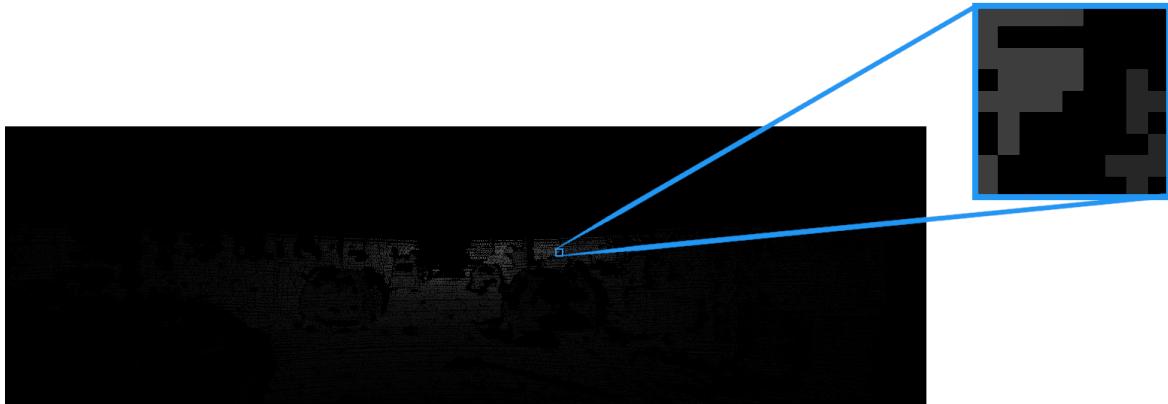


Figura 3.4: Anotación de KITTI y detalle de su carácter disperso para un instante dado.

3.3.3.2. Conjuntos de entrenamiento, validación y evaluación

Para el desarrollo de este trabajo es necesario disponer de un conjunto de entrenamiento con el que ajustar los parámetros de los modelos, un conjunto de validación con el que comprobar que no se está sobre ajustando el modelo al primer conjunto y para elegir la combinación de hiperparámetros óptima, y por último, un conjunto de evaluación (*test*) en el que calcular una serie de métricas que nos aportarán información del rendimiento real de los modelos finales.

El *dataset* de KITTI ya está dividido en entrenamiento y validación, e incluye una descarga adicional con el conjunto de test para el cual no están disponibles públicamente las anotaciones. No obstante, en las publicaciones sobre estimación de profundidad monocular [4, 64, 81, 77] es común encontrar los conjuntos de entrenamiento, validación y evaluación definidos por Eigen et al. [75] (conocidos como el *Eigen split*) que no respetan las particiones originales del *dataset* de KITTI.

Las listas con los archivos que pertenecen a cada uno de las particiones se han descargado desde el repositorio⁹ del trabajo de Godard et al. [64]. En estas listas, hay nombres de archivos que no tienen ninguna anotación asociada, por lo que se eliminan de sus respectivas particiones. La distribución del número de imágenes, así como el número de imágenes eliminadas de cada conjunto se muestran en la Tabla 3.3.

	Eigen split		
	Entrenamiento	Validación	Evaluación
Num. de archivos	45200	1776	697
Imágenes no encontradas	0 (0 %)	0 (0 %)	0 (0 %)
Anotaciones no encontradas	630 (1.39 %)	30 (1.69 %)	45 (6.46 %)
Num. imágenes útiles	44570	1746	652

Tabla 3.3: Distribución de las imágenes y número de imágenes no encontradas en el dataset.

Como comprobación adicional, se han cruzado las listas de archivos descargadas para asegurar que ninguno de los elementos de los conjuntos de entrenamiento y validación se encuentran en el conjunto de evaluación.

⁹<https://github.com/nianticlabs/monodepth2/tree/master/splits> - (*Eigen full*)

3.4. Definición de modelos a entrenar

Uno de los objetivos de este Trabajo Fin de Máster es explorar distintas modificaciones en la arquitectura DPT [4]. Para poder comparar de forma exhaustiva los efectos y la influencia en el rendimiento de cada una de estas modificaciones, se exploran todas las combinaciones posibles de los valores que se van a estudiar. Por lo tanto, no se detiene el entrenamiento de ningún modelo en caso de que su rendimiento sea peor que los modelos ya entrenados, ni se optimiza la búsqueda en función de la influencia de cada modificación en una métrica concreta.

Para definir tantos modelos como combinaciones sean posibles, se toman los conjuntos de valores elegidos para cada una de las modificaciones planteadas y se calcula su producto cartesiano (*grid search*). De esta forma, se obtienen las arquitecturas de los modelos que se entrenarán durante este trabajo y de los que se obtendrán los resultados finales.

3.5. Evaluación

Para la evaluación de los modelos presentados en este trabajo y sus modificaciones, se ha seguido la metodología propuesta en la publicación de Lee et al. [77] para evaluar los resultados en KITTI que consiste en recortar un pequeño marco alrededor de la imagen de salida y crear una máscara para los píxeles que no tienen una profundidad definida en la anotación. En la publicación de DPT [4], también se emplea el mismo procedimiento.

Al usar esta misma metodología se satisfacen dos objetivos: poder reproducir los resultados presentados en dicho artículo con el modelo original, y evaluar las modificaciones introducidas para comparar sus resultados con los del modelo sin modificar.

Como se mencionará más adelante, una de las modificaciones introducidas en este trabajo reduce el tamaño de la imagen en la entrada de las arquitecturas. El resultado de estos modelos, no obstante, se escala a su tamaño original antes de llevar a cabo la evaluación para asegurar que la magnitud de las métricas se ajuste a la del modelo original (con entradas de mayor resolución).

3.5.1. Métricas

Una vez escaladas y enmascaradas las predicciones y las anotaciones, se calculan una serie de valores cuantitativos que permiten comparar y evaluar el rendimiento de los modelos. Las funciones que nos proporcionan estos valores son conocidas como métricas. Dentro del gran número de funciones que permiten evaluar los resultados de los modelos, se han elegido aquellas comúnmente empleadas en los modelos de aprendizaje profundo dedicados a la estimación de profundidad en imágenes monoculares [4, 57, 75, 81, 77, 119, 120, 121].

En las siguientes ecuaciones, d_p representa el valor del mapa de profundidad original (anotación) para cada pixel p , mientras que \hat{d}_p representa el valor de la profundidad estimada por el modelo para cada pixel p . Por otro lado, T denota el número de píxeles con información de profundidad disponibles en la anotación (al ser anotaciones dispersas, las imágenes no tienen información sobre la profundidad en todos los píxeles).

3.5.1.1. Accuracy under a threshold

La primera de estas métricas, el *accuracy under a threshold*, viene dada por la Ecuación (3.14) y cuantifica el porcentaje de píxeles a los que el modelo ha asignado una profundidad cuya relación de escala respecto de su valor real es menor que un determinado umbral. Los valores que se emplean para este umbral son 1,25, 1,25² y 1,25³.

$$\% \text{ de } p \in T : \max\left(\frac{\hat{d}_p}{d_p}, \frac{d_p}{\hat{d}_p}\right) = \delta < \text{umbral} \quad (3.14)$$

3.5.1.2. Mean Absolute Value of the Relative Error (Abs Rel)

Otra métrica usada habitualmente es el promedio del error relativo en todos los píxeles que disponen de valor de profundidad anotada. Para conseguir este error relativo, se calcula el error absoluto y se divide entre el valor real de la profundidad ([Ecuación \(3.15\)](#)).

$$\frac{1}{T} \sum_{p \in T} \frac{|d_p - \hat{d}_p|}{d_p} \quad (3.15)$$

3.5.1.3. Mean Squared Relative Error (Sq Rel)

Similar a la métrica anterior, en este caso el error absoluto se eleva al cuadrado antes de ser dividido entre el valor a estimar y de promediarlo con el resto de píxeles ([Ecuación \(3.16\)](#)). De esta forma, por la naturaleza cuadrática de la fórmula, se le da una mayor importancia a los errores mayores que a los menores.

$$\frac{1}{T} \sum_{p \in T} \frac{(d_p - \hat{d}_p)^2}{d_p} \quad (3.16)$$

3.5.1.4. Linear Root Mean Squared Error (RMSE)

El valor del error cuadrático medio proporciona una medida del promedio de la magnitud de la diferencia entre la profundidad predicha para cada uno de los píxeles y su profundidad real ([Ecuación \(3.17\)](#)). Una característica interesante de esta métrica es que sus unidades coinciden con las de la variable predicha, lo que facilita su interpretación. Como los errores se elevan al cuadrado antes de promediarse, estos tienen una importancia relativa directamente relacionada con su magnitud, es decir, cuanto mayor sea el error, más peso tendrá en el promedio. Es por esto por lo que es especialmente útil si se busca penalizar más los errores más grandes en las predicciones.

$$\sqrt{\frac{1}{T} \sum_{p \in T} (d_p - \hat{d}_p)^2} \quad (3.17)$$

3.5.1.5. Logarithmic Root Mean Squared Error (RMSElog)

Similar a la métrica anterior, en este caso el error cuadrático medio se calcula sobre los logaritmos naturales de las medidas a comparar ([Ecuación \(3.18\)](#)). Al realizar la resta de los logaritmos, la operación es equivalente a calcular el logaritmo de la división del valor de profundidad estimado y el valor de profundidad anotado, restando de esta forma importancia a la escala del error y obteniendo una aproximación al error relativo de las medidas (frente al *RMSE*, que sería una medida del error absoluto). Además, debido al escalado que realizan los logaritmos, los *outliers* pierden importancia, por lo que es una métrica más robusta frente a este tipo de errores puntuales.

Otra característica a destacar de esta métrica es que está sesgada para penalizar aquellos casos en los que el valor predicho es menor que el valor real (subestimación). De esta forma, el error en

dicha situación será mayor que si el valor predicho es mayor que el valor real (sobreestimación) aún cuando la diferencia entre ambos valores sea la misma.

$$\sqrt{\frac{1}{T} \sum_{p \in T} (\ln d_p - \ln \hat{d}_p)^2} \quad (3.18)$$

3.5.1.6. Scale Invariant Logarithmic Error (*SIlog*)

Esta métrica, es la raíz cuadrada de la función de pérdida propuesta por Eigen et al. [75] con $\lambda = 1$ ([Ecuación \(3.19\)](#)). Al fijar el valor de λ en la unidad, se obtiene una medida totalmente independiente de la escala de la salida (demostración matemática disponible en el [Apéndice C](#)). De esta forma, se obtiene una medida de la calidad de los resultados de los modelos ignorando completamente la escala en la que se han producido las predicciones, que como ya se ha comentado es uno de los problemas fundamentales de la estimación de profundidades en imagen monocular.

$$\sqrt{\frac{1}{T} \sum_{p \in T} (\ln \hat{d}_p - \ln d_p)^2 - \left(\frac{1}{T} \sum_{p \in T} \ln \hat{d}_p - \ln d_p \right)^2} * 100 \quad (3.19)$$

3.5.1.7. Mean Logarithmic Error (*Log10*)

Por último, se calculará también el promedio del error (en escala logarítmica) de las profundidades predichas respecto de las profundidades reales siguiendo la [Ecuación \(3.20\)](#).

$$\frac{1}{T} \sum_{p \in T} |\log_{10} d_p - \log_{10} \hat{d}_p| \quad (3.20)$$

3.5.1.8. Velocidad de procesamiento

Además de la calidad de los resultados, es de especial interés en este trabajo obtener medidas relacionadas con la velocidad de procesamiento que pueden alcanzar los modelos. Dentro de las medidas empleadas hay dos tipos: condicionadas por el *hardware* utilizado (Tiempo de inferencia) e independientes del *hardware* (Número de operaciones en coma flotante).

Tiempo de inferencia

Esta medida corresponderá al tiempo que tarda el modelo en procesar **una sola** imagen. Si suponemos que la aplicación de estos modelos es el procesamiento de vídeo de forma online, donde los fotogramas no pueden procesarse en lotes, esta medida es la inversa de los fotogramas por segundo (*FPS*). Como se ha mencionado antes, esta métrica estará sujeta al *hardware* en el que se ejecute, y por lo tanto variará de un equipo a otro.

Número de operaciones en coma flotante (*FLOPs*)

Por último, esta vez independiente del *hardware* en el que se ejecuta el modelo, se empleará el número de operaciones en coma flotante (en inglés, *FLOPs*) que se requieren para procesar una sola entrada de la red como medida de la complejidad y coste computacional de los modelos.

4: Desarrollo y Modificaciones de la Arquitectura

En este capítulo, se presentan los desarrollos principales que se han llevado a cabo durante el trabajo, prestando especial atención a las modificaciones realizadas sobre la arquitectura DPT.

4.1. Cloud

Con el objetivo de reducir el tiempo necesario para entrenar los distintos modelos que se plantean en este trabajo, se ha recurrido al servicio de infraestructura (IaaS - *Infrastructure as a Service*) que ofrece la empresa Google: Google Cloud. Los servicios en la nube (*cloud*), permiten disponer de recursos informáticos de manera flexible, pagando únicamente por aquellos que estén activos. Los proveedores de IaaS, se encargan del mantenimiento y gestión de la infraestructura (redes, almacenamiento, servidores, virtualización), mientras que el usuario se encarga de la gestión del sistema operativo y todo lo que hay por encima.

Dentro de Google Cloud, se han empleado los servicios **Compute Engine**, para disponer de máquinas virtuales y **Cloud Storage**, para crear recursos de almacenamiento (*buckets*).

El flujo de trabajo que se ha seguido ha sido el siguiente ([Figura 4.1](#)):

1. Primero, se ha creado un *bucket* en el que se han dejado disponibles el conjunto de datos empleado durante el entrenamiento, el código necesario para ejecutar el entrenamiento, y una serie de scripts para facilitar la configuración del equipo. Al disponer de estos archivos en la nube, se desacoplan totalmente la configuración de las máquinas virtuales y el ordenador local en el que se lleva a cabo el desarrollo (Equipo 1 en [Tabla 3.1](#)).
2. A continuación, se configura una máquina virtual con el *hardware* elegido¹⁰ (Equipo 2 en [Tabla 3.1](#)). Una vez conectados a esta máquina virtual a través de SSH, se descarga del *bucket* creado el script de configuración (disponible en el [Apéndice B](#)) y se ejecuta. Este script, se encarga de: descargar el resto de archivos disponibles en el *bucket*, instalar los drivers de NVIDIA necesarios para poder usar la GPU de la instancia, instalar Docker y el NVIDIA Container Toolkit, instalar Weights and Biases, y construir la imagen de Docker especificada en el Dockerfile descargado.
3. Una vez configurada la instancia con todos los archivos necesarios en su disco SSD, se crea una imagen de dicha instancia en Google Cloud de forma que sea fácilmente replicable.

¹⁰Para elegir el *hardware* de las máquinas virtuales, se ha elegido la GPU que mayor relación TFLOPS/euro ofrecía para minimizar el coste de los equipos. El resto de características se han elegido de forma que la limitación del equipo sea el procesamiento en GPU.

Posteriormente, se configura desde la consola de Google Cloud el inicio de estas instancias de forma que cada vez que se encienda una (nueva o ya existente), se cree dentro del equipo un contenedor de Docker a partir de la imagen ya construida, y se ejecute en este contenedor el cliente de Weight and Biases para entrenar modelos automáticamente. De esta forma, para añadir una nueva máquina al proceso de entrenamiento de experimentos, solamente hay que crear un nuevo equipo a partir de la imagen preconfigurada.

4. Por último, una vez finalizados los experimentos, se copian a través de SSH los parámetros de los modelos entrenados guardados en cada una de las máquinas virtuales empleadas.

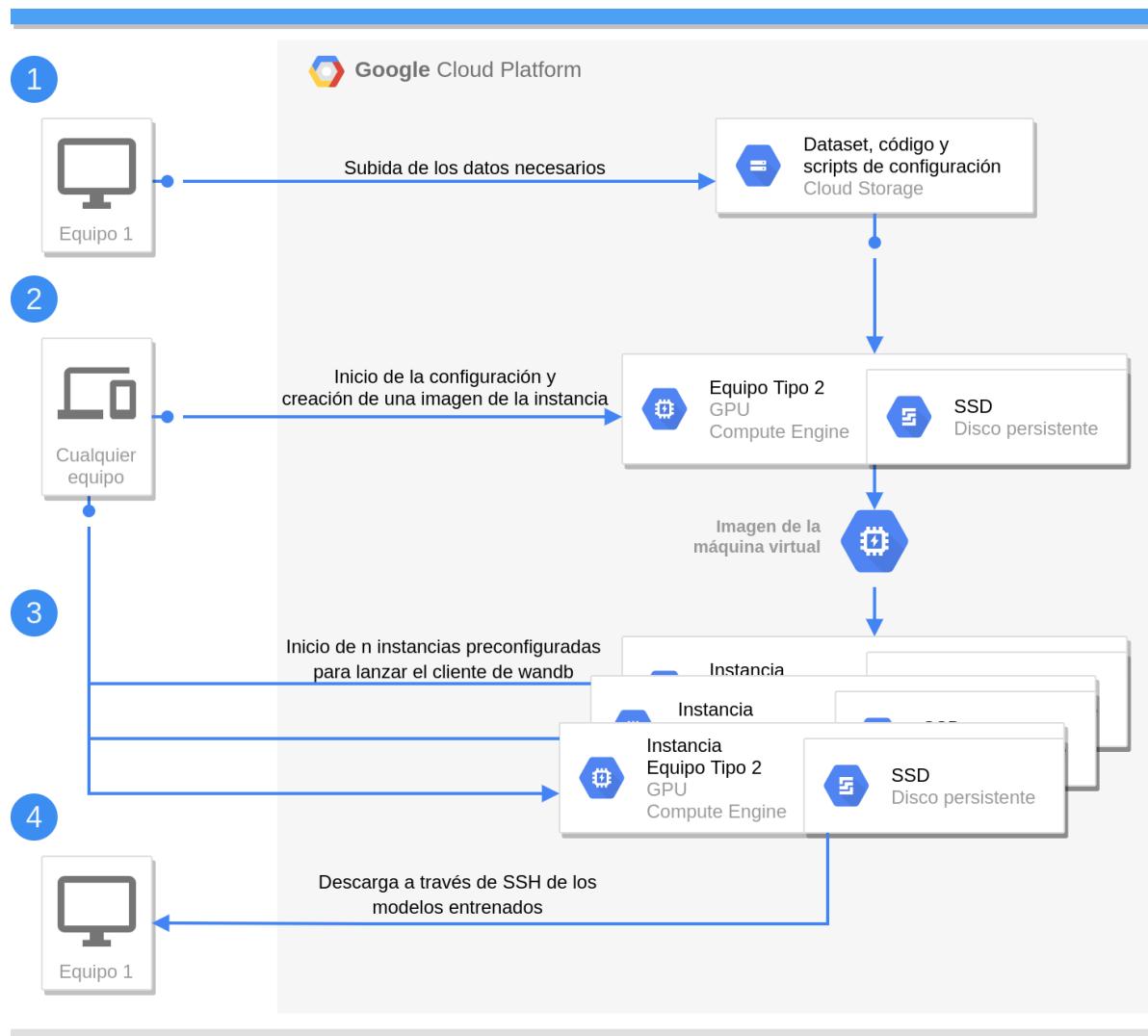


Figura 4.1: Esquema de la configuración llevada a cabo en la nube.

4.2. Warmstart

Para acelerar lo máximo posible la convergencia del modelo durante el entrenamiento, se aprovechan en la medida de lo posible parámetros con sesgos inductivos ya aprendidos, es decir, parámetros de modelos ya entrenados. Más concretamente, los parámetros de los modelos a entrenar se inicializan con los valores de los parámetros del modelo DPT-Hybrid entrenado en MIX6, publicados en el artículo original de *Dense Prediction Transformers* [4].

Además, ya que en este trabajo se estudian diferentes modificaciones de dicho modelo, se emplea el método `load_state_dict()` de la clase `torch.nn.Module` con el parámetro `strict=False` para cargar los parámetros, evitando así que la carga falle si no coinciden una a una las capas en el modelo y las definidas en el archivo que se trata de cargar. En cuanto a las capas/conjuntos de capas que se han modificado y/o añadido, al no disponer del conjunto de datos MIX6 para preentrenarlas, se han inicializado con sus parámetros correspondientes tras ser entrenadas en ImageNet21K.

Esta forma de proceder, sin embargo, conlleva que los modelos con menor porcentaje de su arquitectura modificada partan de una situación inicial ventajosa para la tarea de estimación de profundidad monocular (al estar ya preentrenados en MIX6 en vez de ImageNet).

4.3. Entrenamiento

Pese a que los autores del artículo de DPT [4] han publicado el código del modelo y sus pesos preentrenados en MIX6 y KITTI, a día de hoy no han hecho públicos los *scripts* de entrenamiento empleados. Por esto, ha sido necesario escribir el proceso de entrenamiento así como el **Dataset** de PyTorch con el que leer los datos de KITTI.

Para el **Dataset**, se crea una clase `KITTIDataset`¹¹ que hereda de la clase abstracta base para conjuntos de datos que ofrece PyTorch, `torch.utils.data.Dataset`, y sobreescribe los métodos `__len__()` y `__getitem__()` de forma que estos se adapten a la estructura de directorios y nombres de las imágenes y de sus anotaciones. Al sobreescribir estos métodos, es posible crear un `torch.utils.data.DataLoader` de forma directa, pasando las imágenes y las etiquetas a los modelos aprovechando las herramientas de PyTorch. En el método `__getitem__()`, además, se aplican las transformaciones necesarias a los datos, así como el *Data Augmentation*.

Por *Data Augmentation* se entiende el conjunto de operaciones y transformaciones que se pueden aplicar a los datos para modificar su apariencia. Esto normalmente favorece el aprendizaje y la capacidad de generalización de la red (tiene efecto regularizador y evita el sobreajuste al aumentar el número de ejemplos y la variedad entre ellos). En el entrenamiento llevado a cabo, siguiendo una vez más la metodología de DPT, se ha incluido un **reflejado horizontal aleatorio**, es decir, cada una de las imágenes (junto con las anotaciones) tiene un 50 % de posibilidades de ser reflejada horizontalmente antes de atravesar la red como ejemplo de entrenamiento.

En cuanto al *script* de entrenamiento¹², se tienen en cuenta una serie de factores para acelerar el proceso lo máximo posible:

- **Número de trabajadores en el Dataloader:** Con el objetivo de asegurar que la limitación en la velocidad de entrenamiento sea el procesamiento en la GPU, se cambia el valor del parámetro `num_workers` del constructor del `Dataloader` de entrenamiento a 8. Este parámetro, controla el número de procesos que se lanzan en paralelo para leer los datos del disco y preprocesarlos.
- **pin_memory:** También en el constructor del `Dataloader` es posible activar `pin_memory`, parámetro desactivado por defecto. Este parámetro, acelera la transferencia a la memoria de la GPU de los datos cargados en memoria (RAM) por la CPU [122]. De forma resumida, lo que habilita este parámetro es que la carga de datos se haga en memoria no paginable (*pinned*) a la que la GPU accede directamente, evitando así cargar los datos en una zona de memoria paginable y transferir estos a una *pinned memory* temporal cada vez que la GPU quiere leerlos para transferirlos a su propia memoria.

¹¹<https://github.com/guillessanbri/DPT/blob/v1.0.0-tfm/KITTIDataset.py>

¹²<https://github.com/guillessanbri/DPT/blob/v1.0.0-tfm/train.py>

- **`torch.backends.cudnn.enabled` y `torch.backends.cudnn.benchmark`:** Estas dos opciones, se activan para asegurar, respectivamente, que se use CuDNN en la ejecución del modelo y que se ejecuten al comienzo del *script* distintas implementaciones de algoritmos de convolución para emplear el más rápido en el sistema actual.
- ***Mixed precision:*** Ya mencionado en el [Apartado 2.9](#), pese a que finalmente se ha descartado su uso en el entrenamiento debido a la inestabilidad numérica que introducía y los fallos que ocasionaba, el *script* de entrenamiento incluye la opción de activar el uso de precisión mixta con el escalado pertinente.

Para ajustar los parámetros de los modelos, se ha empleando como optimizador AdamW con una tasa de aprendizaje $lr = 1e - 5$ y parámetros: $\beta_1 = 0,9$, $\beta_2 = 0,999$, $\epsilon = 1e - 8$ y *weight decay* = 0,01. El número de épocas se ha fijado en 20 tras analizar el comportamiento de la pérdida en distintas ejecuciones previas. En cuanto al tamaño de lote usado, se ha utilizado solamente una imagen para cada actualización de parámetros. Esta elección viene motivada por dos razones, la primera, la falta de recursos computacionales para entrenar algunas de las variaciones de los modelos con lotes de más imágenes, y la segunda y más importante, que tras implementar en el *script* de entrenamiento la posibilidad de acumular gradientes¹³, se comprobó que se obtenían mejores resultados con lotes de una sola imagen, probablemente por el efecto regularizador característico de un tamaño de lote tan reducido.

Además, se ha empleado la función `torch.nn.utils.clip_grad_value_`, que se encarga de limitar los valores de los gradientes en función de su valor con un `clip_value` de 0,5, limitando considerablemente las posibilidades de que explotasen los gradientes del modelo y se desestabilice su entrenamiento. No obstante, como precaución y para evitar el desperdicio de recursos, el entrenamiento se detiene en caso de que el valor de la función de pérdida se vuelva $\pm\infty$.

4.3.1. Función de pérdida

La función de pérdida usada durante el proyecto, y por lo tanto implementada en el script de entrenamiento, es la función empleada en la publicación de DPT [4] para ajustar los modelos preentrenados en MIX6 a *datasets* más pequeños. Tal y como indica esta publicación, su función de pérdida se compone de la función de pérdida propuesta por Eigen et al. [75] y de otra función que calcula los gradientes de la profundidad obtenida para penalizar la falta de suavidad en píxeles contiguos, propuesta en el trabajo de Li et al. [82]. No obstante, como el *dataset* empleado en este trabajo es KITTI y sus anotaciones son dispersas, no es posible calcular dichos gradientes en las etiquetas y por lo tanto se elimina esa parte de la función.

De esta forma, la función de pérdida resultante es la definida por la [Ecuación \(4.21\)](#), donde \hat{d}_p es la profundidad obtenida de la red para cada píxel p , d_p es la etiqueta con los valores de profundidad, n es el número de píxeles que tienen un dato de profundidad en la etiqueta correspondiente y λ es un hiperparámetro cuyo valor puede estar en el rango $[0, 1]$ y controla la influencia de la escala, ya que con $\lambda = 0$ el segundo término se anula (quedando la distancia de cuadrados en espacio logarítmico), y con $\lambda = 1$ la función de pérdida es invariante a la escala. La demostración de la invariancia a la escala es equivalente a la de la métrica *SIlog* ([Apéndice C](#)), ya que dicha métrica es la raíz cuadrada de esta función de pérdida. Para el entrenamiento de los modelos, λ se ha fijado en 0,5.

¹³La acumulación de gradientes (*gradient accumulation*) es una técnica que promedia las pérdidas en distintos lotes para, aún sin poder aprovechar la mejora de rendimiento de un tamaño de lote mayor, conseguir que la actualización de los parámetros sea matemáticamente equivalente (con alguna limitación como la imposibilidad de usar *batch normalization*) a usar un tamaño de lote mayor.

$$L(\hat{d}, d) = \frac{1}{n} \sum_p (\ln \hat{d}_p - \ln d_p)^2 - \frac{\lambda}{n^2} \left(\sum_p (\ln \hat{d}_p - \ln d_p) \right)^2 \quad (4.21)$$

4.4. Modificaciones introducidas en DPT

En esta sección se describen las distintas modificaciones y desarrollos que se han introducido en la arquitectura original de DPT a lo largo del proyecto, y que por lo tanto, conforman las distintas variantes del modelo que se entrenarán y evaluarán.

4.4.1. Reducción de tamaño de la entrada

Para acelerar DPT lo primero que se modificó fue el tamaño de la entrada, buscando estudiar la flexibilidad de la arquitectura para aprender a inferir profundidad a partir de imágenes con menor resolución. Los resultados de la publicación original calculan la profundidad en el conjunto de evaluación de KITTI con las imágenes en su tamaño original, 1216x352. Para reducir el consumo de memoria del modelo durante su entrenamiento, así como acelerar entrenamiento e inferencia, se han añadido dos operaciones de cambio de tamaño: una al principio de la red que reduce el tamaño de las imágenes a 640x192 píxeles, y otra al final de la red que reescalada la salida al tamaño original.

4.4.2. Capas de atención eficiente

Tal y como se ha visto en el [Apartado 2.6.3](#) de Marco Teórico y Estado del Arte, existen numerosas técnicas para reducir la complejidad computacional y requisitos de memoria de los mecanismos de atención característicos de los *transformers*. Sin embargo, estas técnicas ofrecen mayores incrementos en el rendimiento cuanto mayor es la longitud de la cadena de la entrada. Por lo tanto (especialmente tras reducir el tamaño de las imágenes) no se va a apreciar un incremento especialmente substancial en la velocidad de inferencia por modificar estos mecanismos. Aún así, se decide modificar el mecanismo de atención con el objetivo de estudiar la capacidad de estas nuevas técnicas para igualar los resultados de los mecanismos originales. En concreto, se compara el mecanismo de atención del *Perfomer* con la atención estándar.

Para poder llevar a cabo los experimentos relacionados con este aspecto de la arquitectura y poder entrenar los modelos necesarios, se modifica el código de DPT de forma que se puedan seleccionar distintos mecanismos de atención, entre ellos la implementación del mecanismo de atención del *Perfomer* desarrollada por Phil Wang [\[99\]](#). Entre los distintos mecanismos de atención eficiente, se ha elegido el del *Perfomer* debido a que tiene una complejidad $O(n)$ y también debido a que, según apuntan los autores en su publicación, es posible reutilizar los parámetros de las matrices de pesos W^V , W^K y W^Q de los mecanismos de atención estándar para obtener las matrices Q , K y V (hace falta un pequeño ajuste, pero no es necesario entrenar desde cero).

4.4.3. Número de cabezas

En la publicación *Are Sixteen Heads Really Better than One?* [\[123\]](#), se plantea y prueba la idea de que el número de cabezas en los bloques de atención está sobredimensionado, siendo posible eliminar un gran porcentaje de las cabezas sin afectar al rendimiento de forma significativa. Esta influencia en el rendimiento, se reduce aún más cuando se reentrena el modelo después de reducir el número de cabezas. Además, los autores señalan que reducir el número de cabezas degradaría el rendimiento de forma más significativa en los bloques de *Cross-Attention* ([Apartado 2.6.2](#)) con elementos del *encoder* y *decoder* que en los bloques de *Self-Attention*.

En el caso de DPT, al emplear como *encoder* un ViT, no existen bloques de *Cross-Attention*, si no que son todos de *Self-Attention* (recordemos que el ViT solo se compone de la parte del *encoder* del *transformer* original). Por lo tanto, se plantea también como una de las modificaciones del proyecto estudiar la influencia del número de cabezas en la velocidad de inferencia y los resultados, adaptando el código del modelo para que sea posible cambiar la cantidad de cabezas en los bloques de atención (tanto de atención estándar como de atención eficiente). Los valores con los que se experimenta para este hiperparámetro son 1, 12 (el número de cabezas que tiene DPT por defecto) y 24.

4.4.4. Hooks del transformer y bloques de atención posteriores

En la arquitectura de DPT ([Apartado 2.8.5](#)), la transferencia de información desde el *encoder* al *decoder* tiene lugar a través de cuatro ganchos o *hooks* que toman las salidas de ciertas capas para que pasen a ser las entradas que se fusionan en el *decoder*. En la publicación original, si bien es cierto que se valoran diferentes tamaños de *Vision Transformers*, no se presenta ningún experimento modificando los bloques de atención de los cuales se cogen las salidas. Más concretamente, el modelo DPT-Hybrid coloca dos de estos *hooks* en el *backbone* convolucional y los otros dos en los bloques de atención número 8 y número 11 (empezando en cero). Por lo tanto, la imagen de entrada tiene que atravesar todos los bloques de atención del ViT-Hybrid para conseguir la salida del último bloque de atención.

Los valores que se han probado en este trabajo para estos *hooks* del ViT, es decir, los bloques de atención cuya salida se ha seleccionado como entrada del *decoder*, son: bloques número 0 y 1, bloques número 2 y 5, y por último, bloques número 8 y 11 (siendo estos últimos los de la publicación original).

Al modificar las capas del transformer de las que se cogen las activaciones para pasarlas a la etapa de fusión convolucional, se abre la posibilidad de eliminar aquellos bloques de atención que ya no se usan. Por lo tanto, se ha modificado el código de la arquitectura para incluir dos cambios en el momento de instanciar el modelo: el primero, capar la inferencia para que la ejecución del ViT se detenga cuando llega al bloque de atención más alto requerido por los *hooks*, y el segundo, eliminar las capas superiores no empleadas. De esta forma, además de acelerar el entrenamiento e inferencia del modelo, se reduce su tamaño considerablemente, tanto a la hora de almacenar sus parámetros como a la hora de cargarlo en memoria para desplegarlo en una aplicación real.

En la [Figura 4.2](#) se puede apreciar la modificación del ViT empleado en DPT: a la izquierda está el *encoder* tal y como se encuentra en el trabajo de DPT, con los *hooks* en los bloques 8 y 11, sin eliminar ninguno de los parámetros del *transformer*, y con la inferencia recorriendo los 12 bloques de atención; a la derecha, por otro lado, está una de las opciones valoradas para este hiperparámetro de la arquitectura, donde los *hooks* se sitúan en la salida de los bloques 0 y 1 del ViT. De esta forma, en el ejemplo, los parámetros a partir del segundo bloque de atención se eliminarían ya que la inferencia del modelo solo llega hasta este segundo bloque.

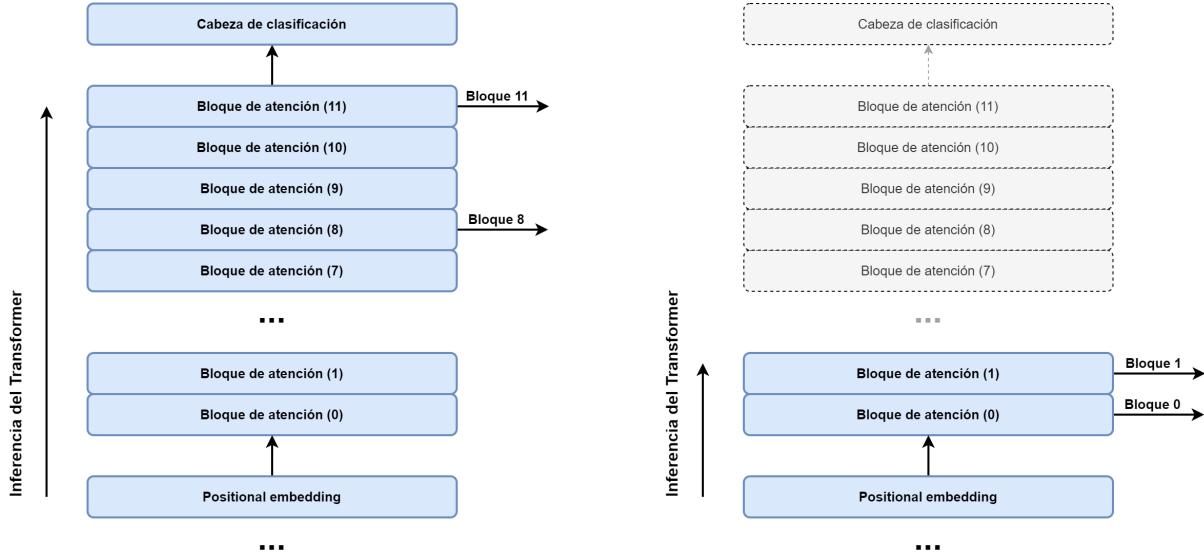


Figura 4.2: Cambio en los *hooks* y en el número de bloques de atención.

4.4.5. Backbone convolucional

La última modificación de la arquitectura de DPT estudiada en este proyecto es la del *backbone* convolucional del ViT-Hybrid, encargado de extraer los mapas de características empleados para obtener los *tokens* que pasan a los bloques de atención.

En el modulo propuesto en la publicación original, se elige como *backbone* una ResNet50. De esta red, se extraen las activaciones en los bloques 0 y 1 para pasarlas al *decoder*. Por otro lado, la salida de la última capa de la ResNet50, es decir, la entrada de los bloques de atención, tiene forma $[n, c, h, w]$, donde: n es el número de imágenes en el lote (*batch size*), c es el número de canales (en este caso mapas de características), y h y w la altura y anchura de los mapas de características. En el caso de este *backbone*, la salida está compuesta por $c = 1024$ mapas de características de dimensiones iguales a las de la imagen de entrada divididas 4 veces entre 2.

Después de la ResNet, hay una capa de proyección que no es más que una capa convolucional con *kernels* de tamaño 1×1 y zancada 1×1 , con 1024 canales de entrada y 768 canales de salida. Este tipo de convoluciones 1×1 , realmente se componen de 786 kernels de $1 \times 1 \times 1024$, por lo que al convolucionar cada uno de ellos con la entrada, se obtienen 768 mapas de características del mismo tamaño que los de la entrada. Después, los 768 mapas de características resultantes se apllanan en tensores de forma $[n, 768, \frac{h}{8} \times \frac{w}{8}]$ y se transponen de forma que la entrada sea de tipo $[n, t, 768]$, donde 768 es el tamaño de los *tokens* de los bloques de atención y t el número de *tokens* extraídos de la imagen. De esta forma, a mayor tamaño de imagen mayor será el número de *tokens* (secuencia más larga, aumenta el coste de los bloques de atención), pero la dimensión de estos permanece constante.

A modo de ejemplo, supongamos una entrada de una sola imagen de tamaño 384×384 : la salida de la ResNet50 tendrá la forma $[1, 1024, 24, 24]$. Esta salida atraviesa la convolución 1×1 de proyección y pasa a tener forma $[1, 768, 24, 24]$. Estos mapas de características se apllanan en un tensor $[1, 768, 576]$ que se traspone para obtener la forma $[1, 576, 768]$, que equivale a 576 *tokens* de dimensión 768. Una vez llegados a este punto, el tensor está listo para pasar a los bloques de atención del *transformer*.

Buscando una alternativa que ofrezca resultados similares con un menor coste computacional, se adapta e incluye en las pruebas la arquitectura EfficientNet-B0 como *backbone* convolucional de DPT. Esta arquitectura, proporciona una salida de forma $[n, c, h, w]$ (manteniendo la nomenclatura anterior) donde el número de mapas de características es ahora 1280 y la altura

y anchura se corresponden con las de las imágenes de entrada divididas 5 veces entre 2. Es decir: $[n, 1280, \frac{h}{16}, \frac{w}{16}]$. Para sustituir el *backbone* convolucional del ViT-Hybrid sin tener que cambiar el tamaño de todas las capas de atención (y así aprovechar los pesos preentrenados), se sustituye la capa de proyección mencionada en el párrafo superior, que recordemos era una capa convolucional con kernels de tamaño 1×1 por una capa de convolución transpuesta. Esta capa de convolución transpuesta, cumple dos funciones fundamentales: la primera, proyectar los 1280 mapas de características en 768 gracias al número de *kernels* empleados; y la segunda, al tener *kernels* de tamaño 2×2 y una zancada también de 2×2 , conseguir que las dimensiones de los mapas de características se multipliquen exactamente por 2 en ambas dimensiones, convirtiendo la entrada $[n, 1280, \frac{h}{16}, \frac{w}{16}]$ en $[n, 768, \frac{h}{8}, \frac{w}{8}]$. Estas dimensiones, son las que espera la etapa de *embedding* que aplana los mapas de características y transpone el tensor para obtener la entrada de los bloques de atención, quedando a su salida un tensor con forma $[n, t, 768]$ donde t vuelve a ser el número de *tokens* que pasan al *transformer*, cada uno de dimensión 768.

Dado que no se disponen de los parámetros ajustados en MIX6 para EfficientNet-B0, se ha modificado también el *script* del modelo para cargar en su inicialización los parámetros ajustados en ImageNet21K (proporcionados por la librería PyTorch Image Models [98], de donde también se ha utilizado la implementación de EfficientNet-B0) correspondientes a este nuevo módulo.

4.5. Pruebas de cuantificación

Además de las modificaciones comentadas, se han probado exhaustivamente distintos *frameworks* de optimización y cuantificación. Más concretamente, se han probado los frameworks de optimización TensorRT (TRT) de NVIDIA y TF Lite, de TensorFlow. Para estas pruebas, se han utilizado tanto los modelos originales de PyTorch (en TensorRT) como los modelos convertidos a ONNX (ecosistema que permite la interoperabilidad entre *frameworks* de aprendizaje profundo). Además de con TRT y TF Lite, se ha tratado de cuantificar el modelo tanto estáticamente como dinámicamente empleando el cuantificador de PyTorch.

En el caso de los dos primeros *frameworks* de optimización, no se ha conseguido ejecutar de forma exitosa la optimización de los modelos. La razón de los fallos obtenidos es que algunas de las capas de la arquitectura de DPT-Hybrid no están soportadas por la conversión de modelos, es decir, sería necesario ampliar las capacidades de las herramientas para implementar la optimización de dichas capas. Por otro lado, con el cuantificador de PyTorch tampoco se han conseguido resultados satisfactorios, en este caso, debido principalmente a la extracción de salidas intermedias del *encoder* para pasárlas al *decoder*, que ocasiona inconsistencias entre las operaciones que se llevan a cabo como enteros de 8 bits y como números en coma flotante de 32 bits.

Las dificultades encontradas al usar estos paquetes de *software* son solucionables ampliando las funcionalidades de las herramientas. No obstante, este desarrollo no es trivial y excede el alcance de este Trabajo Fin de Máster. Por lo tanto, se propone su desarrollo como una posible línea de trabajo futuro.

5: Resultados

En este capítulo se exponen los resultados obtenidos en el conjunto de validación tras entrenar las arquitecturas con las modificaciones planteadas. Tal y como se plantearon en el [Apartado 3.4](#), los experimentos han consistido en entrenar 36 modelos definidos por el producto cartesiano de las opciones: (ResNet50, EfficientNet-B0) como *backbone*, (Atención estándar, Atención *Performer*) como mecanismo de atención, (1, 12, 24) como número de cabezas de atención, y ([0, 1], [2, 5], [8, 11]) como bloques de atención de los que se toma la salida para que sea la entrada del *decoder*. En este capítulo, además de las métricas que miden la exactitud de las salidas, se presentan también métricas de velocidad de inferencia. Para medir este último grupo de variables, tras comprobar que los resultados numéricos en la salida de los modelos eran idénticos, se emplea precisión mixta para aumentar la velocidad lo máximo posible ([Figura 5.1](#)).

5.1. Resultados cuantitativos

5.1.1. Reducción del tamaño de la entrada

Dado que el entrenamiento de los modelos - por limitaciones materiales y temporales - se ha llevado a cabo reduciendo el tamaño de la entrada, no es posible comparar los resultados

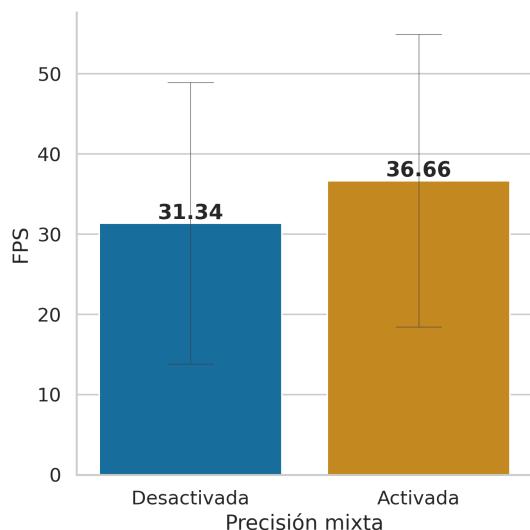


Figura 5.1: FPS promedios de los modelos en función de la precisión durante la inferencia. Las barras grises representan la desviación estándar de las medidas.

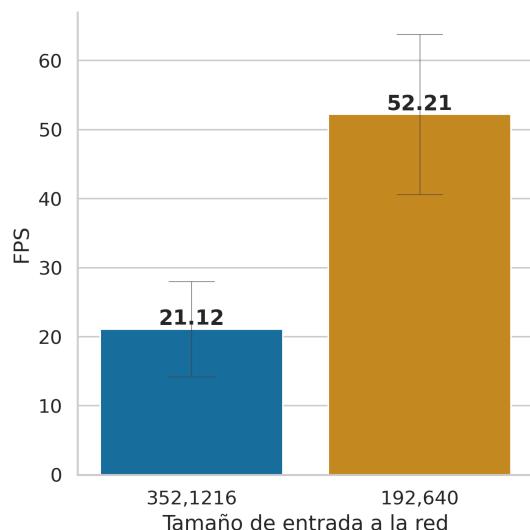


Figura 5.2: FPS medios de los modelos en función del tamaño de su entrada. Las barras grises representan la desviación estándar de las medidas.

obtenidos para cada una de las modificaciones de la arquitectura con su modelo entrenado con las imágenes en su tamaño original.

Sin embargo, la medida de la velocidad de inferencia sí que es posible obtenerla independientemente de que los pesos correspondan al modelo entrenado o no. En la [Figura 5.2](#) se puede apreciar la diferencia en función del tamaño de entrada de los FPS promedio de todas las modificaciones realizadas sobre el modelo DPT. El valor de la izquierda, corresponde al tamaño con el que se evalúa KITTI en la publicación original [4], mientras que el de la derecha es la reducción de tamaño establecida en este trabajo. Dado que este cambio es significativo y es, en términos de magnitud, el que mayor aceleración media conlleva ($\times 2,47$), en las siguientes Figuras que representen los FPS en función de los otros cambios introducidos en el trabajo se presentarán los datos tanto con el tamaño de entrada original como con el tamaño de entrada reducido para poder así valorar también la mejora de rendimiento que conllevaría la modificación si no se hubiese reducido el tamaño de la entrada.

5.1.2. Cambio del backbone convolucional

El cambio del *backbone* convolucional usado para extraer los mapas de características, como era de esperar, también influye en la velocidad de inferencia de la red. En la [Figura 5.3](#), es posible ver el incremento en los FPS promedio entre los modelos con ResNet50 y los modelos con EfficientNet-B0.

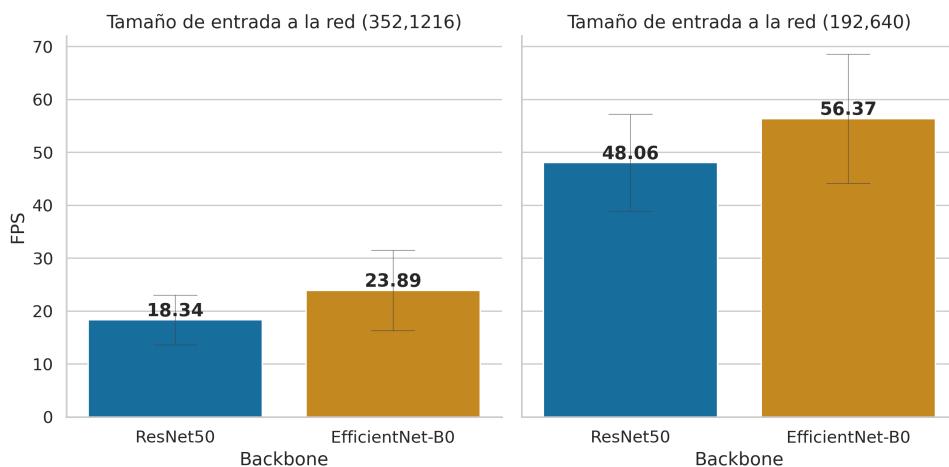


Figura 5.3: FPS medios de los modelos en función del *backbone* utilizado y del tamaño de la entrada. Las barras grises representan la desviación estándar de las medidas.

A pesar de que el incremento de velocidad es positivo, existe un problema asociado a la modificación del *backbone*. De todas las modificaciones desarrolladas y estudiadas en este trabajo, el cambio de *backbone* domina de forma muy considerable el deterioro de la calidad de los resultados, esto se puede ver claramente, por ejemplo, en la comparación del error logarítmico invariante a la escala (SIlog) disponible en la [Figura 5.4](#).

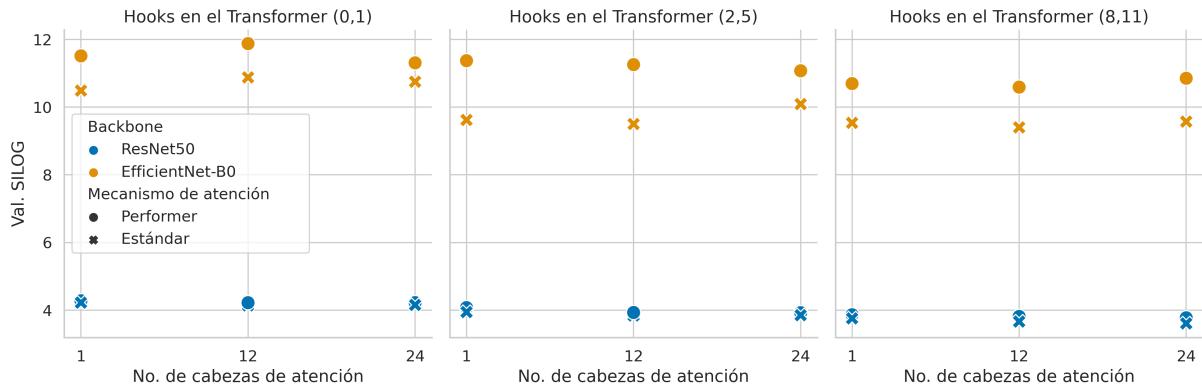


Figura 5.4: SIlog obtenido en el conjunto de validación para cada uno de los 36 modelos entrenados. Más bajo es mejor.

Con el objetivo de buscar la causa de esta diferencia tan grande del error, se estudian las curvas de aprendizaje de entrenamiento y validación. En estas curvas (Figura 5.5), se puede observar como en comparación con el modelo equivalente con ResNet50, el modelo que tiene como *backbone* la EfficientNet-B0 está sobreajustando (*overfitting*) sus parámetros al conjunto de entrenamiento, ya que la diferencia entre el error en el conjunto de entrenamiento y el error en el conjunto de validación es mucho mayor en el caso del modelo con EfficientNet.

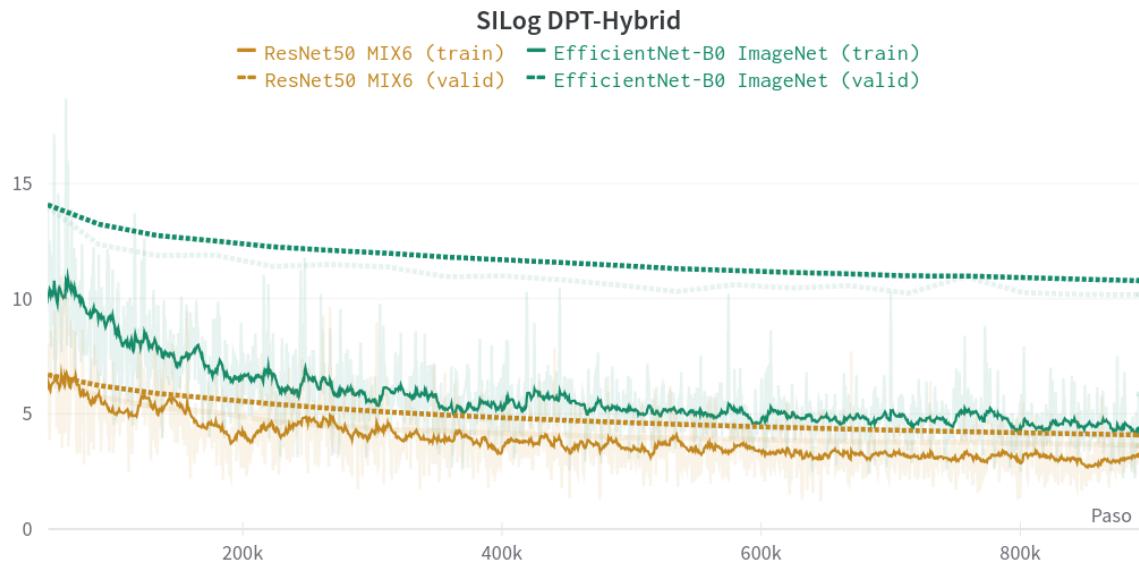


Figura 5.5: Comparación de las curvas de aprendizaje del SIlog durante el entrenamiento y validación de dos modelos con diferentes *backbones* e inicializaciones de pesos. Valores suavizados con una media exponencial ponderada para facilitar su visualización.

Dado que los parámetros de EfficientNet-B0 se han inicializado con parámetros entrenados en ImageNet y no en MIX6 (*dataset* de profundidad utilizado para preentrenar las ResNet50), se plantea un experimento adicional para descartar que esta falta de preentrenamiento especializado sea la causa del *overfitting*. Para llevar a cabo este experimento, se extraen los parámetros de un ViT-Hybrid preentrenado en ImageNet y se utilizan para inicializar los pesos de un DPT-Hybrid con ResNet50. De esta forma, la ResNet50 parte de un estado similar al de la EfficientNet-B0 (preentrenamiento en ImageNet). En la Figura 5.6, se pueden apreciar los resultados en el conjunto de validación de estos tres modelos y se comprueba como el dominio del preentrenamiento no influye en el sobreajuste del *backbone* convolucional.

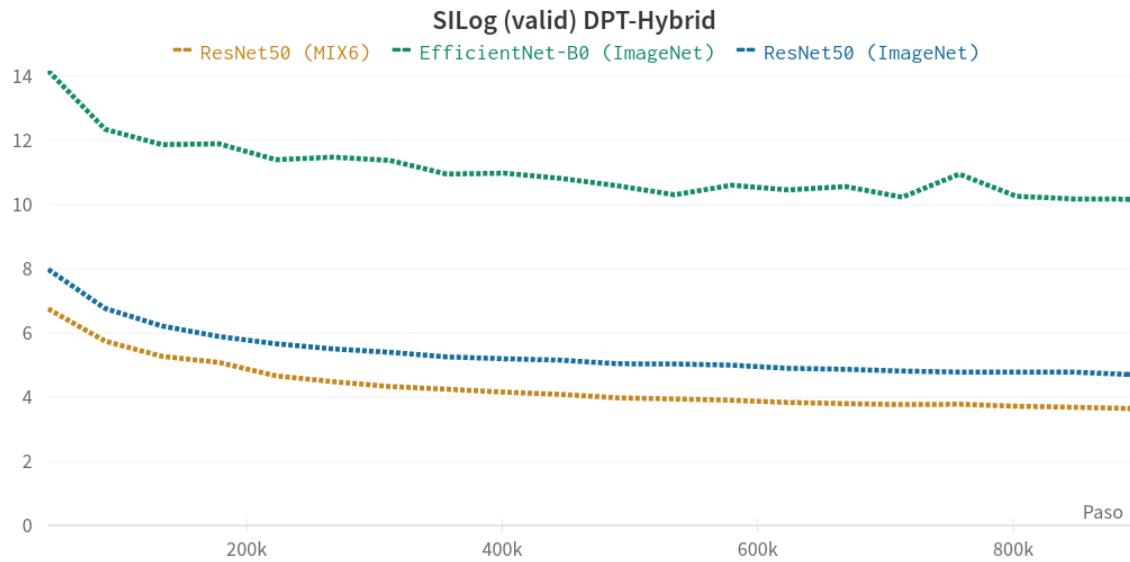


Figura 5.6: Comparación del SIlog en el conjunto de validación durante el entrenamiento en tres modelos con diferentes inicializaciones de pesos.

5.1.3. Número de cabezas

En la Figura 5.7, equivalente a la Figura 5.4 pero separada en función del *backbone*, es posible ver como la diferencia en los resultados según el número de cabezas en los bloques de atención es relativamente pequeña, especialmente en el caso del mecanismo de atención estándar.

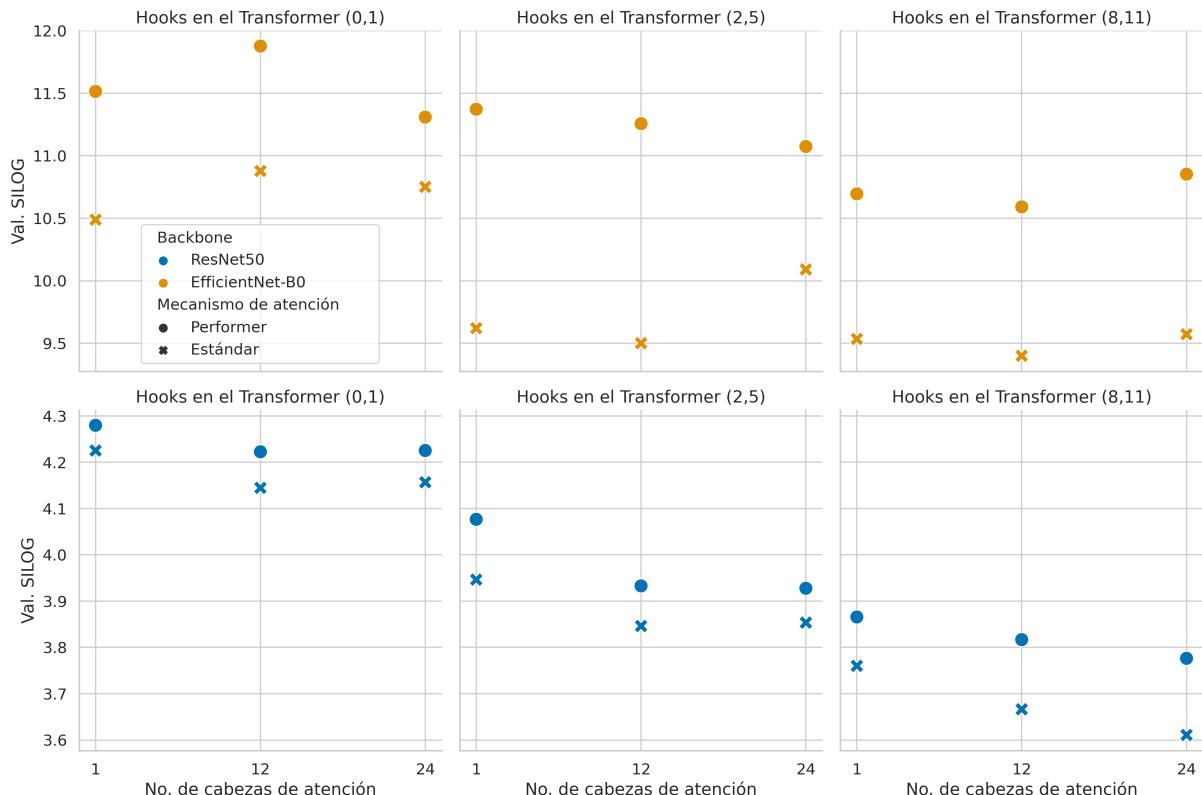


Figura 5.7: SIlog obtenido en el conjunto de validación para cada uno de los 36 modelos entrenados con distintos ejes en función del *backbone*. Más bajo es mejor.

De forma similar, en el caso de las métricas de velocidad de inferencia ([Figura 5.8](#)), sobretodo en el caso de las imágenes reducidas, no se encuentra un aumento de la velocidad especialmente significativo.

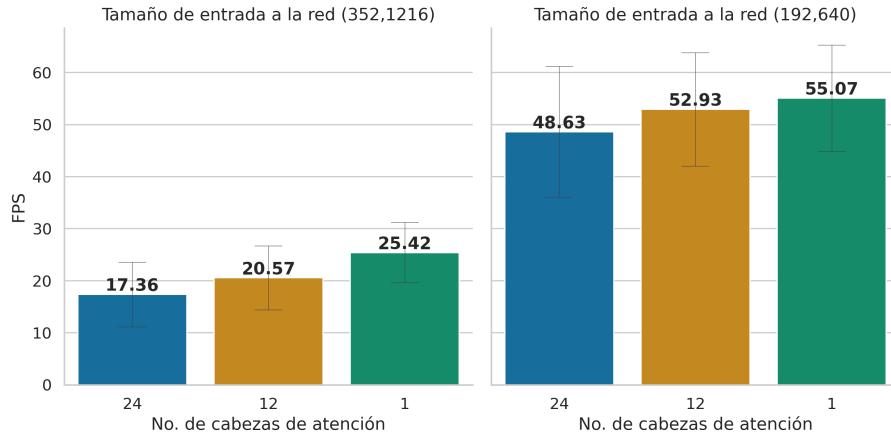


Figura 5.8: FPS medios de los modelos en función del número de cabezas utilizado y del tamaño de la entrada. Las barras grises representan la desviación estándar de las medidas.

5.1.4. Capas de atención eficiente

Tal y como se ha visto en la [Figura 5.7](#), el cambio del mecanismo de atención eficiente también influye negativamente en la calidad de los resultados. Dado que el incremento de la métrica de error es relativamente pequeño, esto podría ser totalmente aceptable si no fuese por los resultados de las métricas de velocidad ([Figura 5.9](#)), donde se puede observar que los modelos con el mecanismo de atención eficiente (*Performer*) son en realidad más lentos cuando el tamaño de la entrada es reducido.

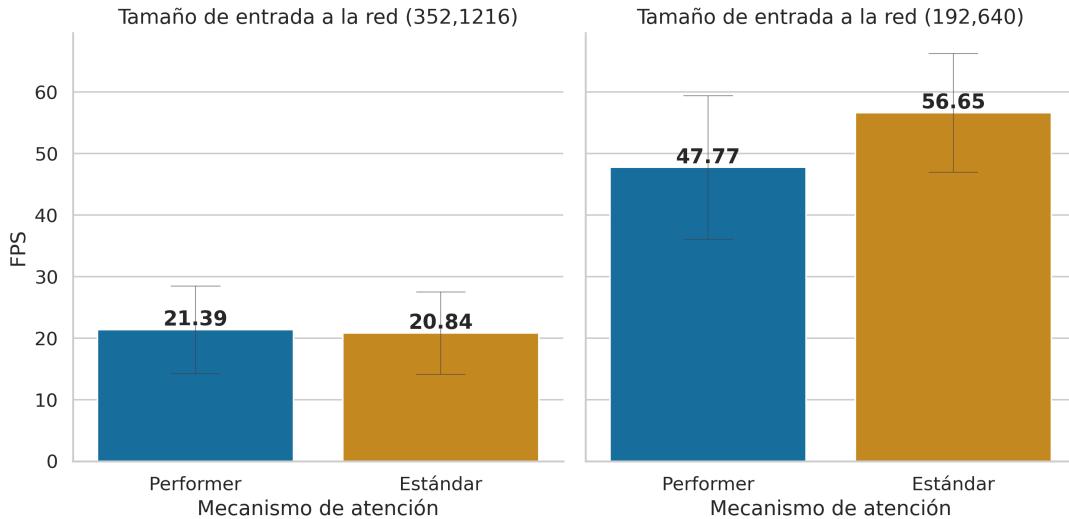


Figura 5.9: FPS medios de los modelos en función del mecanismo de atención utilizado y del tamaño de la entrada. Las barras grises representan la desviación estándar de las medidas.

Esto, se debe a que, pese a que la complejidad del mecanismo de atención eficiente es $O(n)$ y la del mecanismo de atención estándar es $O(n^2)$, existe un *overhead* asociado al mecanismo de

atención eficiente que hace que la ejecución para secuencia de elementos cortas (como es el caso de la imagen reducida) sea más lento que con la atención estándar. En la [Figura 5.10](#), donde se muestran los resultados de otro experimento donde se incrementó la longitud de la secuencia de entrada progresivamente para medir el tiempo de inferencia de un bloque de atención, se puede apreciar este comportamiento.

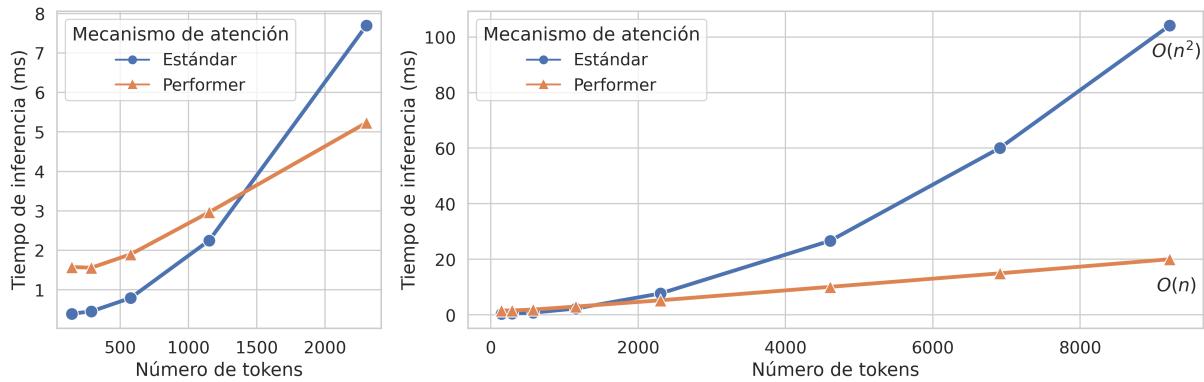


Figura 5.10: Comparación de la complejidad y el tiempo de inferencia en un bloque de atención usando el mecanismo de atención estándar y el mecanismo eficiente del *Performer*. Detalle de los tiempos de inferencia para entradas con un número de *tokens* reducido a la izquierda.

5.1.5. Cambio en los hooks del transformer y eliminación de las capas de atención posteriores

La última modificación de la arquitectura estudiada, el cambio en los *hooks*, es decir, la elección de los bloques de atención de los cuales se toman las salidas como entradas del *encoder* es, después del cambio de *backbone*, la que más afecta a la calidad de los resultados ([Figura 5.7](#)). Por otro lado, el incremento en la velocidad de inferencia asociado a este cambio sí que es considerable ([Figura 5.11](#)) llegando casi a duplicar los FPS en los modelos con la entrada a resolución original y aumentando en más de un 50 % los FPS cuando el tamaño de la entrada se ha reducido.

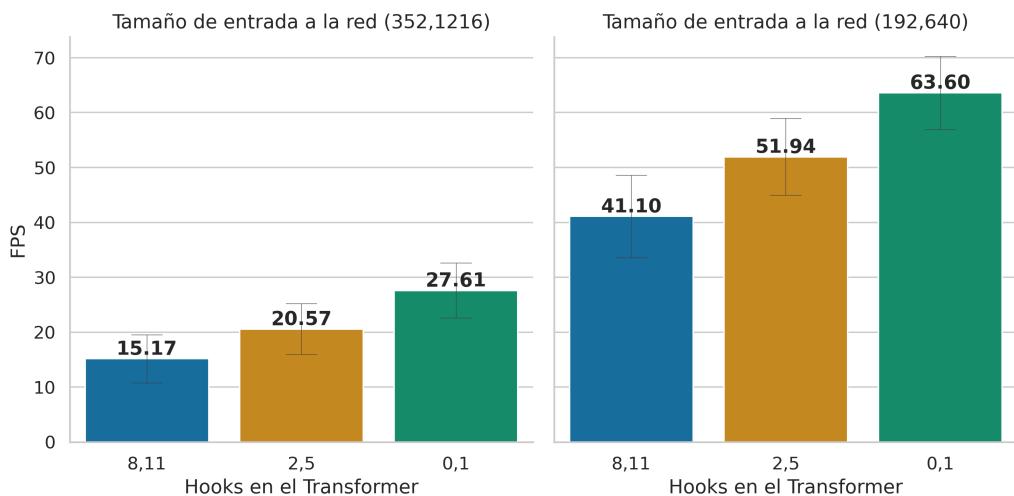


Figura 5.11: FPS medios de los modelos en función de los *hooks* utilizados y del tamaño de la entrada. Las barras grises representan la desviación estándar de las medidas.

5.2. Tamaño de los modelos y número de operaciones

El tamaño de los modelos también se ve afectado por las modificaciones del modelo que se han llevado a cabo en este proyecto. En la Figura 5.12 se puede ver el tamaño en *megabytes* (MB) que ocupan los modelos. En esta gráfica, puede apreciarse como la mayor diferencia viene dada por los bloques donde se sitúan los *hooks* ya que al elegir los bloques iniciales del *transformer* es posible eliminar todos los parámetros correspondientes a los bloques de atención posteriores. Cabe destacar también de estos resultados cómo la diferencia de tamaño entre los modelos en función del *backbone* convolucional es relativamente pequeña, y la forma en que el número de cabezas afecta al tamaño del modelo en función del mecanismo de atención empleado.

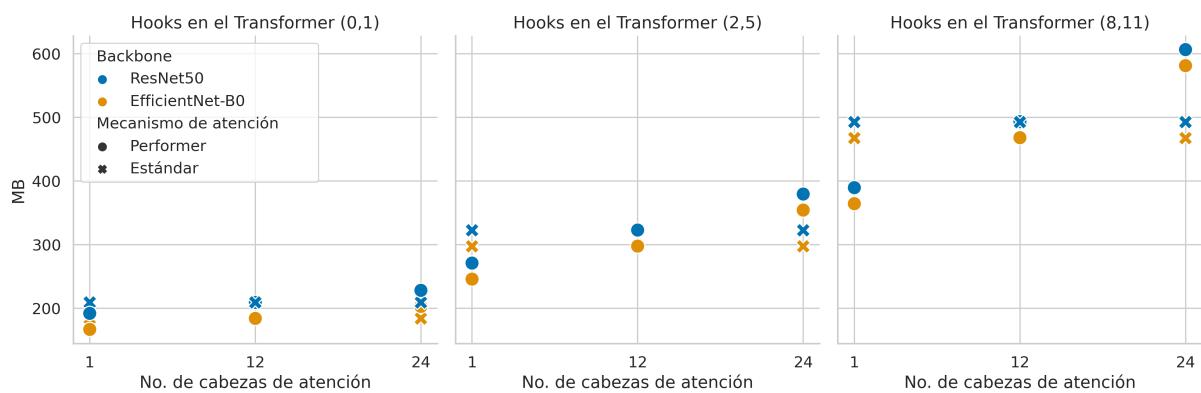


Figura 5.12: Tamaño en memoria de los modelos en función de su configuración.

La tendencia de los tamaños de los modelos en función de sus características es posible verla también en las medidas de Giga FLOPs (número de operaciones en coma flotante requeridas para inferir la salida con una sola entrada) presentadas en la Figura 5.13. No obstante, una diferencia interesante entre estas dos Figuras es que en el número de operaciones necesarias para obtener la salida del modelo sí que influye en mayor medida la elección del *backbone* convolucional, es decir, se puede comprobar que para un número de parámetros similar, EfficientNet-B0 lleva a cabo un número de operaciones menor que ResNet50.

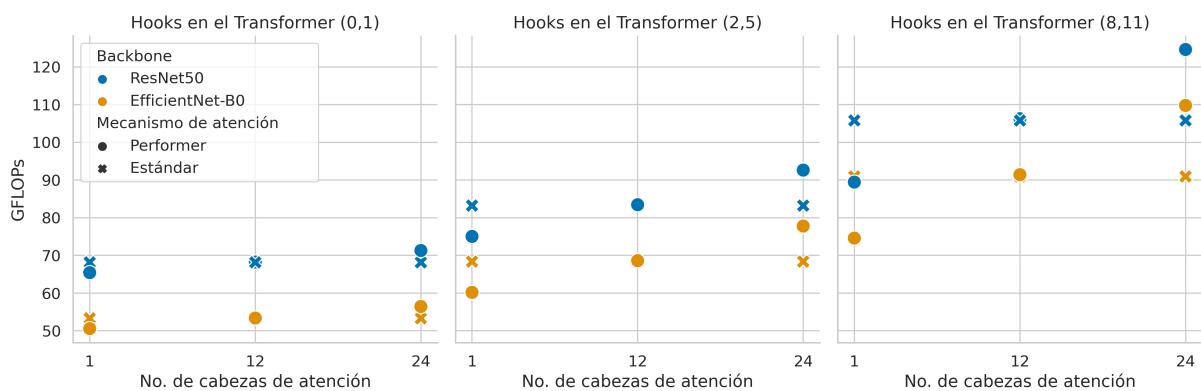


Figura 5.13: Número de operaciones en coma flotante necesarias para inferir la profundidad en una imagen en función de la configuración del modelo.

5.3. Resultados cualitativos

En esta sección, se presentan los resultados en el conjunto de evaluación (*test*) de distintos modelos en función de sus modificaciones, con el objetivo de comparar visualmente y apreciar las diferencias entre las salidas que generan las distintas arquitecturas. Los valores de las imágenes generadas por el modelo solamente ocupan una pequeña parte de la escala de grises, por lo tanto, para facilitar la apreciación de detalles, las imágenes mostradas a continuación han sido previamente normalizadas de forma que ocupen todo el rango de grises.

En la primera comparación de resultados ([Figura 5.14](#)), se encuentran las salidas del modelo original DPT-Hybrid publicado en [4] y el modelo equivalente entrenado durante este trabajo. Es decir, la diferencia entre los dos modelos es el tamaño de las imágenes con las que se ha entrenado y el *script* de entrenamiento empleado (como recordatorio, la imagen se reduce antes de ser procesada, pero después se vuelve a escalar a su tamaño original en la salida). En cuanto a las dos salidas, si bien es cierto que en la salida correspondiente a la entrada reducida (c) se pueden apreciar unos bordes menos suavizados, también es posible observar como la salida es considerablemente similar.

Por otro lado, en la [Figura 5.15](#), donde se comparan las salidas del modelo original entrenado con imágenes reducidas y el modelo equivalente pero con EfficientNet-B0 como *backbone*, sí que se puede observar un deterioro de los resultados que se corresponde con la diferencia en las métricas del [Apartado 5.1.2](#), por ejemplo, en el pilar del puente mostrado en el detalle.



Figura 5.14: Resultados en función de la resolución de entrada. (a) Entrada del modelo y detalle. (b) Salida del modelo proporcionado en el repositorio de DPT [4] (sin reducir la entrada). (c) Salida del modelo equivalente a DPT con el tamaño de entrada reducido.



Figura 5.15: Resultados en función del *backbone* empleado. (a) Entrada del modelo y detalle. (b) Salida del modelo equivalente a DPT con el tamaño de entrada reducido (ResNet50). (c) Salida del modelo equivalente a DPT con el tamaño de entrada reducido y EfficientNet-B0 como *backbone*.

En la [Figura 5.16](#), se encuentran las salidas de tres modelos que difieren en el número de cabezas de sus bloques de atención. En estas imágenes, sin embargo, es difícil apreciar diferencias, ya que como se ha visto en el [Apartado 5.1.3](#), la diferencia en las métricas en función de este parámetro es relativamente pequeña. A modo de detalle, se señala en el recorte de la derecha de esta Figura cómo la salida del modelo con una sola cabeza (c), estima incorrectamente la profundidad en el reflejo del lateral de la camioneta, mientras que los otros dos modelos sí que lo identifican correctamente.



Figura 5.16: Resultados en función del número de cabezas de atención del modelo. (a) Entrada del modelo y detalle. (b) Salida del modelo equivalente a DPT con el tamaño de entrada reducido (12 cabezas de atención). (c) Salida del modelo equivalente a DPT con el tamaño de entrada reducido y 1 cabeza de atención. (d) Salida del modelo equivalente a DPT con el tamaño de entrada reducido y 24 cabezas de atención.

En el caso de la [Figura 5.17](#), es posible comparar las salidas de dos modelos cuya única diferencia es el mecanismo de atención empleado. En este caso, de forma similar al cambio del número de cabezas, es difícil observar diferencias notables. A modo de detalle, y en contra de lo que indican las métricas en el conjunto de validación, es posible observar como el modelo con la atención estándar (b) predice una profundidad errónea para el coche del reflejo del cristal de un edificio, mientras que el modelo con el mecanismo de atención del *Performer* si que identifica correctamente la pared.



Figura 5.17: Resultados en función de los *hooks* empleados. (a) Entrada del modelo y detalle. (b) Salida del modelo equivalente a DPT con el tamaño de entrada reducido (atención estándar). (c) Salida del modelo equivalente a DPT con el tamaño de entrada reducido y atención tipo *Performer*.

Por último, en la Figura 5.18, lo que cambia en los modelos son los *hooks* en los bloques de atención, y por lo tanto está directamente relacionada con el Apartado 5.1.5. En estas salidas, sí que es posible visualizar como los resultados son peores en algunos detalles, sobre todo en objetos de menor tamaño en la imagen como son las señales de tráfico señaladas en el detalle de la derecha o los carteles más alejados.



Figura 5.18: Resultados en función de los *hooks* empleados. (a) Entrada del modelo y detalle. (b) Salida del modelo equivalente a DPT con el tamaño de entrada reducido (*hooks* en bloques [8, 11]). (c) Salida del modelo equivalente a DPT con el tamaño de entrada reducido y *hooks* en los bloques [2, 5]. (d) Salida del modelo equivalente a DPT con el tamaño de entrada reducido y *hooks* en los bloques [0, 1].

6: Discusión

En este capítulo, se comentan una serie de observaciones extraídas a partir de los resultados presentados en la sección anterior. Después, se valoran los modelos obtenidos realizando una comparación con otras arquitecturas del estado del arte.

El cambio del *backbone* original (ResNet50) por una red EfficientNet-B0 conlleva una reducción del tamaño del modelo ([Figura 5.12](#)) y del número de GFLOPs ([Figura 5.13](#)), acelerando este último factor el procesamiento del modelo considerablemente. No obstante, en los distintos experimentos llevados a cabo, se ha podido detectar un claro *overfitting* del modelo al conjunto de datos empleado durante el entrenamiento. Esto nos indica que la EfficientNet-B0, pese a tener un menor número de parámetros, tiene una mayor capacidad que la ResNet50. Sin embargo, esta mayor capacidad no está aprendiendo a extraer características útiles de las imágenes, si no que está memorizando los ejemplos vistos durante el entrenamiento. Por esta razón, el uso de este nuevo *backbone* queda totalmente desaconsejado hasta que se disponga de los resultados tras repetir el entrenamiento de estos modelos aplicando técnicas de regularización mayores para reducir el *overfitting*.

En cuanto al número de cabezas de atención en cada capa, los resultados obtenidos refuerzan la idea presentada en la publicación de Michel et al. [[123](#)], ya que es posible observar cómo la modificación del número de cabezas apenas influye en las métricas sobre el conjunto de validación ([Figura 5.7](#)), especialmente cuando se usa el mecanismo de atención estándar. Por otro lado, la influencia en la velocidad de procesamiento de este cambio tampoco es especialmente grande ([Figura 5.8](#)), pero aún así se considera beneficiosa para el objetivo de acelerar la arquitectura inicial.

Por otro lado, el mecanismo de atención del *Performer* se ha probado inadecuado para este tipo de modelo, ya que al haber reducido el tamaño de la entrada, la cadena de *tokens* extraídos no es lo suficientemente grande para que el *overhead* de esta operación sea despreciable frente a la complejidad de la atención estándar ([Figura 5.10](#)). Esto, en cierta medida, era de esperar, ya que este tipo de mecanismos han sido diseñados para trabajar con largas cadenas de texto con miles y miles de *tokens*. No obstante, también se ha podido comprobar en los experimentos llevados a cabo como la disminución de la calidad de los resultados es bastante reducida ([Figura 5.7](#)), por lo que este tipo de mecanismos siguen siendo prometedores para su uso con imágenes de muy alta resolución donde reducir o trocear la entrada para su procesamiento no sea una opción.

La última de las modificaciones, el cambio de los *hooks* que extraen las activaciones de los bloques intermedios para pasarlos al *decoder*, ha sido la más exitosa, ya que cumple una triple función con muy buenos resultados: no reducir drásticamente las métricas de los resultados (en especial en el conjunto de evaluación final, como se comentará más adelante), aumentar sustancialmente el número de imágenes que puede procesar el modelo en un segundo (FPS) ([Figura 5.11](#)), y por último, reducir el tamaño del modelo ([Figura 5.12](#)) al eliminar los bloques de atención posteriores.

En vista de estas observaciones, se propone como un buen equilibrio entre calidad de los resultados y rendimiento modificar la arquitectura original con las siguientes modificaciones: reducción del tamaño de entrada; cambio de los bloques de atención para solo usar el primero y segundo (*hooks* en [0, 1]), y reducción del número de cabezas a solamente 1. Es decir, se descartan el cambio de *backbone* por su evidente sobreajuste al conjunto de entrenamiento (por lo menos mientras no se repita el estudio regularizando en mayor medida el entrenamiento) y el cambio de mecanismo de atención, ya que las entradas son demasiado pequeñas para poder aprovechar la reducción de complejidad computacional que lleva asociada.

El modelo obtenido con dichas modificaciones, no obstante, es importante compararlo con los modelos del estado del arte actuales, tanto convolucionales como basados en *transformers*. En la [Tabla 6.1](#), se puede encontrar esta comparación. En concreto, se presentan los resultados de los modelos: GLPDepth (GLP) [79], Adabins (ADA) [57], Big To Small (BTS) [77] y DPT-Hybrid (DPT-H) [4]. Además, se presentan las métricas de dos modelos de los entrenados durante este trabajo: el modelo equivalente a DPT-Hybrid con solamente la entrada reducida (DPT-H-r) y el modelo con la entrada reducida, los *hooks* en los bloques [0, 1] y solamente una cabeza de atención en cada bloque (A).

En la [Tabla 6.1](#), hay dos modelos, Adabins y GLPDepth, cuyos resultados no coinciden exactamente con los publicados en la evaluación de KITTI por sus respectivos autores. Esto se debe a que se ha usado el mismo *script* en todos los modelos para calcular las métricas y así poder comparar los resultados de forma justa. Para elegir los modelos mostrados en la [Tabla 6.1](#), se han tomado 4 de los 5 modelos con mejores resultados ordenados por su AbsRel en el *benchmark* del *Eigen Split* de KITTI disponible en *Papers With Code*¹⁴.

Modelo	δ_1 (↑)	AbsRel (↓)	SILog (↓)	FPS (↑)	MB (↓)
GLP	0.960	0.061	8.60	20.3	282
ADA	0.957	0.062	8.72	10.9	897
BTS	0.954	0.061	9.03	33.6	595
DPT-H	0.959	0.062	8.29	13.0	492
DPT-H-r	0.937 (↓ 2.3 %)	0.074 (↑ 19.3 %)	10.20 (↑ 23.0 %)	43.4 (↑ 232.8 %)	492 (↓ 0.0 %)
A	0.938 (↓ 2.2 %)	0.074 (↑ 19.3 %)	10.19 (↑ 22.9 %)	61.1 (↑ 368.8 %)	209 (↓ 57.5 %)

Tabla 6.1: Comparación de distintos modelos del estado del arte. Resultados medidos en el conjunto de test del *Eigen Split* de KITTI. Tiempos de inferencia calculados con precisión mixta en el Equipo 1 de la [Tabla 3.1](#). Se muestran los incrementos/decrementos de las métricas de las modificaciones de DPT-Hybrid respecto de la arquitectura original.

Al comparar estos resultados, es posible observar varias cosas. Existe una diferencia notable en las métricas de Error Absoluto Relativo (AbsRel) y Error Logarítmico invariante a la escala (SILog) entre DPT-Hybrid y las modificaciones introducidas (DPT-H-r y A). Aunque porcentualmente las diferencias no son pequeñas (19,3 % y 22,9 %, respectivamente), hay que tener en cuenta que los modelos resultantes ocuparían el 8º puesto de 37 modelos en el *benchmark* antes citado de *Papers With Code*, es decir, siguen siendo modelos muy competitivos en el campo de la estimación de profundidades monocular. Además de esto, el incremento y el decremento en la velocidad de inferencia y en el tamaño del modelo, respectivamente, es mucho mayor que la pérdida de calidad en los resultados, haciendo que el modelo (A) duplique la velocidad de inferencia del segundo modelo más rápido de la [Tabla 6.1](#) y multiplique casi por cinco la velocidad de inferencia del DPT-Hybrid original.

¹⁴Disponible en <https://paperswithcode.com/sota/monocular-depth-estimation-on-kitti-eigen>. (Consultado el 9 de febrero de 2022)

Por otro lado, cabe destacar que el modelo DPT-H-r (es decir, el modelo equivalente a DPT-Hybrid con solo el tamaño de la entrada reducido) y el modelo A, que modifica los *hooks* y el número de cabezas de atención, han pasado de tener una cierta diferencia en las métricas en el conjunto de validación (donde la salida no se aumentaba al tamaño original de la entrada) a ofrecer resultados prácticamente idénticos. Aquí se puede apreciar también que la mayoría del error introducido en las modificaciones viene dado por la reducción del tamaño de entrada, por lo que sería interesante estudiar el comportamiento de DPT-Hybrid **sin reducir** el tamaño de entrada, pero sí cambiando los *hooks* y el número de cabezas de atención.

Por último, si bien es cierto que la velocidad de inferencia ha aumentado muy considerablemente, no hay que olvidar que aún existe la posibilidad de cuantificar el modelo si se solucionan los problemas planteados en dicha adaptación. Es decir, aún existe la posibilidad de multiplicar el número de imágenes procesadas por segundo sin apenas afectar al rendimiento del modelo.

Este último punto, sin embargo, tiene una doble lectura, y es que los problemas que han surgido durante la cuantificación se han debido principalmente al uso de los *hooks* para trasladar información de las salidas intermedias del *encoder* al *decoder*. Por lo tanto, a pesar de que se puede ampliar el software de cuantificación para soportar este tipo de operaciones, un modelo más sencillo con operaciones más empleadas como puede ser un modelo convolucional, probablemente conlleve menos trabajo para ser cuantificado y por lo tanto acelerado.

7: Conclusiones y Lineas Futuras

En este Trabajo Fin de Máster, se ha estudiado: (1) El campo de la estimación de profundidad, haciendo especial hincapié en el uso de imágenes monoculares y las técnicas de aprendizaje automático supervisado propuestas para este problema; (2) las arquitecturas basadas en *transformers*; y (3) las distintas técnicas dirigidas a acelerar el entrenamiento y la inferencia de los modelos, tanto generales como específicas para arquitecturas concretas. Sobre esta base de conocimiento, se han definido una serie de modificaciones en una de las arquitecturas del estado del arte actual en la estimación de profundidad a partir de imágenes monoculares. Estas modificaciones se han implementado para entrenar los modelos resultantes (con un equipo propio y recursos en la nube) en un conjunto de datos enfocado a la conducción autónoma, KITTI. Una vez los modelos han sido entrenados, se ha llevado a cabo un estudio del grado de influencia de cada una de las modificaciones planteadas en distintas métricas relacionadas con la calidad de los resultados, la velocidad de inferencia, o el tamaño de los modelos modificados.

En base a este estudio de los resultados, se ha seleccionado una configuración que se considera valiosa al tener un equilibrio entre aumento de la velocidad de inferencia, reducción del tamaño del modelo y pérdida de la calidad de los resultados. Junto a esta memoria, se publica el código necesario para entrenar y usar estos modelos, así como sus parámetros tras el entrenamiento, con el objetivo de que sea posible continuar ampliando este trabajo y sus resultados.

En cuanto a las líneas por las que continuar con este proyecto, queda aún pendiente adaptar el modelo de DPT y las modificaciones planteadas para funcionar en *hardware* empotrado, para lo que sería necesario solucionar las limitaciones encontradas relacionadas con la cuantificación (es decir, desarrollar o ampliar los *frameworks* necesarios para cuantificar estos modelos). Por otro lado, sería también interesante entrenar las modificaciones sin reducir el tamaño de entrada, o recopilar los *datasets* que componen MIX6 para entrenar parte de los modelos propuestos, llevando a cabo un estudio exhaustivo de la influencia del preentrenamiento empleando un *dataset* con imágenes de profundidad en comparación con un *dataset* más general como es ImageNet. Además, visto el sobreajuste de los modelos cuando se emplea EfficientNet como *backbone* convolucional, añadir una fuerte regularización al entrenamiento de este grupo de modelos y estudiar sus resultados sería beneficioso. Por último, y con un carácter más aplicado, se podría continuar con este Trabajo Fin de Máster implementando los modelos presentados como parte de una aplicación completa (SLAM monocular, conducción autónoma, reconstrucción 3D, realidad aumentada, etc.) incluso llegando a aprovechar los bloques de atención eficientes con imágenes mucho mayores (por ejemplo, con imágenes satelitales o fotografías de muy alta resolución).

Apéndices

Apéndice A

Planificación del proyecto

Este proyecto de investigación se desarrolla sobre la base del trabajo llevado a cabo durante la asignatura I+D+i en Informática del Máster en Ingeniería Informática de la Universidad de Valladolid, donde se realizó una revisión del Estado del Arte relativo a la estimación de profundidad monocular. Por lo tanto, la planificación planteada no incluye esta primera etapa.

Las tareas planificadas (T) para este Trabajo Fin de Máster, se han agrupado en paquetes de trabajo (PT) en base a su similitud. En la [Figura A.1](#) se puede encontrar tanto la lista de tareas planteadas, como la asignación de recursos temporales a cada una de ellas.

Actividad	Horas														
	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150
PT0: Documentación del trabajo															
T0.1: Redacción de documentación técnica y recogida de notas															
T0.2: Redacción de memoria del Trabajo Fin de Máster															
T0.3: Lectura de documentación y artículos relacionados															
PT1: Familiarización con las herramientas a emplear															
T1.1: Adaptación al entorno de PyTorch															
T1.2: Desarrollo de modelos y conceptos avanzados con PyTorch															
PT2: Definición de la metodología de evaluación															
T2.1: Selección de benchmark y definición de métricas															
T2.2: Descarga y exploración del conjunto de datos															
T2.3: Desarrollo de código para trabajar con el dataset															
PT3: Modificación de la arquitectura															
T3.1: Estudio del repositorio de código de la publicación															
T3.2: Desarrollo e implementación de las modificaciones															
T3.3: Validación de las modificaciones															
PT4: Entrenamiento de los modelos *															
T4.1: Desarrollo de código necesario															
T4.2: Puesta en marcha del entrenamiento (local) *															
T4.3: Desarrollo de la infraestructura en cloud															
T4.4: Puesta en marcha del entrenamiento (cloud) *															
PT5: Optimización de modelos															
T5.1: Pruebas de cuantificación															
PT6: Evaluación de los modelos entrenados															
T6.1: Toma de medidas en el conjunto de pruebas															
T6.2: Estudio y comparación de los resultados obtenidos															

Figura A.1: Cronograma del proyecto.

(*) Cabe mencionar que el entrenamiento de los modelos ha sido un proceso especialmente largo, y por lo tanto se ha intentado en la medida de lo posible aprovechar este tiempo para la redacción de la documentación y las pruebas de cuantificación de modelos.

Apéndice B

Documentación

B.1. Introducción

El siguiente apéndice se compone de tres apartados principales: El primero presenta la estructura de directorios en la que se organiza el proyecto; el segundo apartado documenta cómo descargar los datos empleados para entrenar y evaluar los modelos, así como las instrucciones para instalar los componentes *software* requeridos por los programas desarrollados en este trabajo; por último, en el tercer apartado, se especifican los comandos necesarios para ejecutar los *scripts* disponibles.

B.2. Estructura de directorios

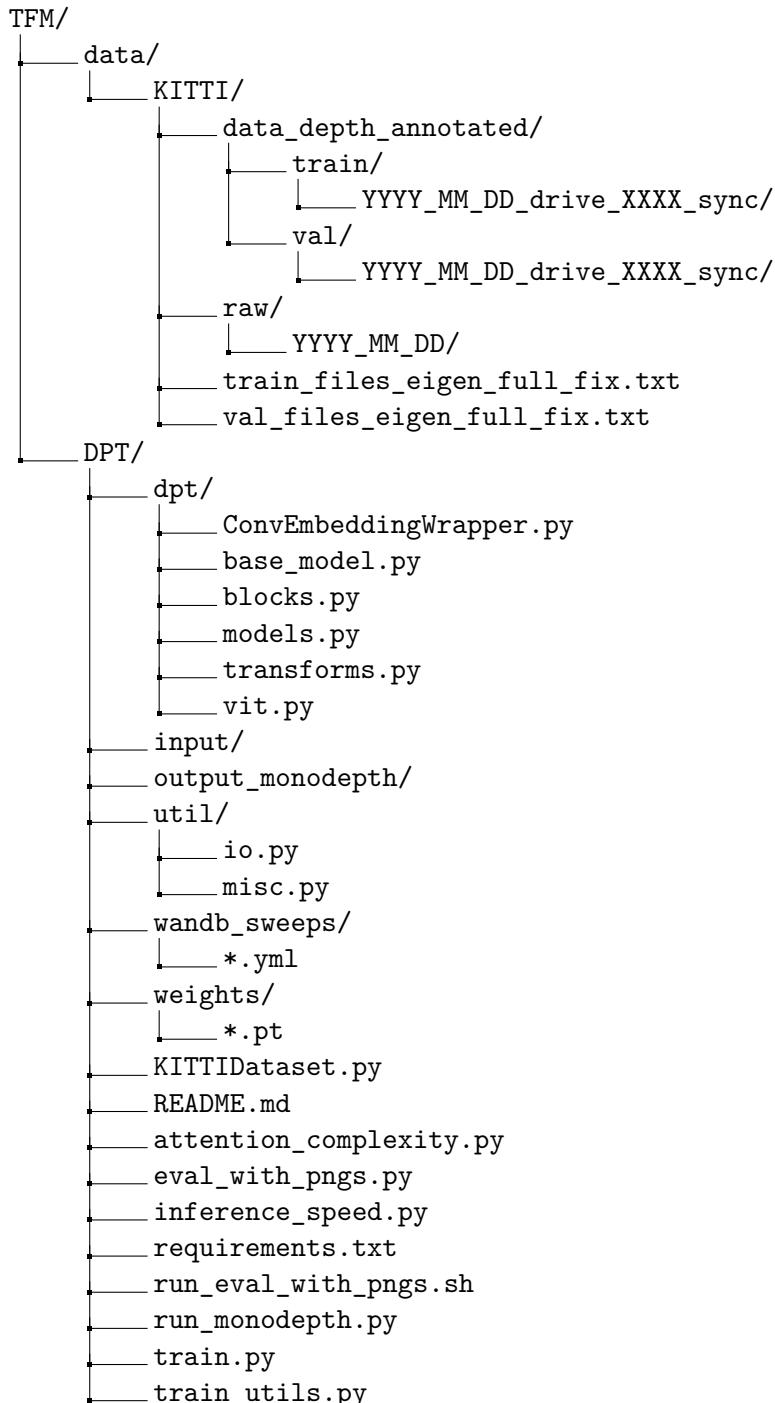
A continuación se encuentra la estructura de directorios y ficheros de este Trabajo Fin de Máster. Esta estructura se divide en dos directorios principales: *data/*, que contiene el *dataset* de KITTI; y *DPT/*, que es un *fork* del repositorio de la publicación original [4] disponible en <https://github.com/guillesanbri/DPT/tree/v1.0.0-tfm>.

En este segundo directorio, se encuentran a su vez los subdirectorios:

dpt/, que agrupa los *scripts* que definen la arquitectura, modificados para incluir los distintos cambios presentados en este trabajo; *input/* y *output_monodepth/* contienen respectivamente, las entradas de los *scripts* de inferencia y sus salidas; *util/* incluye archivos con funciones de utilidad general; *wandb_sweeps/* agrupa los distintos ficheros YAML que definen las búsquedas de hiperparámetros que se han llevado a cabo empleando wandb; por último, en *weights/* se encuentran los ficheros con los parámetros entrenados de los distintos modelos.

En cuanto a los ficheros situados de *DPT/*:

KITTIDataset.py, define el *torch.utils.data.Dataset* adaptado para cargar las imágenes y anotaciones de KITTI; *attention_complexity.py* y *inference_speed.py* ejecutan, respectivamente, distintas capas y distintos modelos para obtener métricas sobre su velocidad, tamaño, y número de operaciones; *eval_with_pngs.py* y *run_eval_with_pngs_.sh* calculan métricas del rendimiento de los modelos; *run_monodepth.py* lleva a cabo la inferencia de profundidad; y por último, *train.py* y *train_utils.py* gestionan el entrenamiento de modelos.



B.3. Descarga de datos e instalación de *software*

A continuación se encuentran los pasos necesarios para descargar los datos empleados durante el proyecto e instalar el *software* de los entornos de desarrollo. **Parte de estas instrucciones son específicas para equipos con sistema operativo Ubuntu 20.04 LTS**, por lo que se proporcionan enlaces con información relevante para otras distribuciones. Además, se presupone que el equipo en el que se va a llevar a cabo el proceso de instalación tiene una tarjeta gráfica NVIDIA con los controladores gráficos (*drivers*) correspondientes correctamente instalados.

B.3.1. Descarga de datos

El *dataset* empleado para entrenar y evaluar los distintos modelos es el conjunto de datos de KITTI para estimación de profundidad en imágenes monoculares. Este *dataset* se puede descargar a través de su página web, tanto los datos en bruto (*synced+rectified data*) http://www.cvlibs.net/datasets/kitti/raw_data.php, como las anotaciones (*annotated depth maps*) http://www.cvlibs.net/datasets/kitti/eval_depth.php?benchmark=depth_prediction.

Los ficheros que definen los conjuntos de entrenamiento y de evaluación se encuentran disponibles en el siguiente repositorio https://github.com/nianticlabs/monodepth2/tree/master/splits/eigen_full.

Por último, las imágenes del conjunto de *test* (*kitti_eval_dataset.zip*), se han descargado agrupadas desde <https://github.com/isl-org/DPT/blob/main/EVALUATION.md>.

B.3.2. Docker Engine

Para utilizar los contenedores de Docker, es necesario instalar el Docker Engine. Estas instrucciones son para sistemas con Ubuntu. En el siguiente enlace (<https://docs.docker.com/engine/install/>), hay disponibles instrucciones para otras distribuciones.

1. Eliminamos versiones anteriores de Docker (en caso de existir):

```
$ sudo apt-get remove docker docker-engine docker.io containerd runc
```

2. Instalamos paquetes necesarios para usar un repositorio a través de HTTPS:

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates curl \
gnupg lsb-release
```

3. Añadimos la clave GPG de Docker:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg \
--dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

4. Configuramos el repositorio de Docker:

```
$ echo \
"deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

5. Instalamos Docker:

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

6. Por último, comprobamos que la instalación se haya llevado a cabo correctamente:

```
$ sudo docker run hello-world
```

Docker ya está instalado, pero es necesario ejecutarlo como superusuario. Para remediar esto (opcional), es posible crear un grupo "docker" y añadir nuestro usuario. Al hacer esto **pueden aparecer riesgos de seguridad**. Se puede encontrar más información en:

- <https://docs.docker.com/engine/install/linux-postinstall/>

- <https://docs.docker.com/engine/security/#docker-daemon-attack-surface>

1. Añadimos un grupo "docker":

```
$ sudo groupadd docker
```

2. Añadimos nuestro usuario al grupo creado:

```
$ sudo usermod -aG docker $USER
```

3. Salimos y volvemos a iniciar la sesión para re-evaluar la pertenencia a grupos. En sistemas Linux también se puede ejecutar `newgrp docker`.

4. Ahora es posible ejecutar el contenedor `hello-world` sin usar `sudo`.

```
$ docker run hello-world
```

Por otro lado, es posible configurar el servicio de Docker para que se inicie automáticamente al encender el sistema a través de `systemctl`:

- `$ sudo systemctl enable docker.service`
- `$ sudo systemctl enable containerd.service`

B.3.3. NVIDIA Container Toolkit

Una vez instalado el Docker Engine, es necesario instalar el NVIDIA Container Toolkit, que envuelve la instalación anterior y traslada las primitivas de CUDA desde el CUDA instalado dentro de los contenedores al driver de la GPU en el equipo donde se está ejecutando el contenedor. Estas instrucciones son para equipos con distribuciones Ubuntu y Debian. Para los pasos correspondientes en otras distribuciones, consultar la siguiente url: <https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html#getting-started>

1. Configuramos el repositorio y copiamos la clave GPG:

```
$ distribution=$( . /etc/os-release; echo $ID$VERSION_ID ) \
&& curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | \
sudo apt-key add - \
&& curl -s -L \
https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list | \
sudo tee /etc/apt/sources.list.d/nvidia-docker.list
```

2. Instalamos el paquete correspondiente:

```
$ sudo apt-get update
$ sudo apt-get install -y nvidia-docker2
```

3. Para terminar la instalación, reiniciamos el servicio de Docker:

```
$ sudo systemctl restart docker
```

4. Para comprobar que la instalación es correcta, podemos ejecutar un contenedor de una imagen con CUDA instalado:

```
$ docker run --rm --gpus all nvidia/cuda:11.0-base nvidia-smi
```

NVIDIA-SMI 510.47.03			Driver Version: 510.47.03		CUDA Version: 11.6		
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	MIG M.
<hr/>							
0	NVIDIA GeForce ...	Off	00000000:08:00.0	On			N/A
0%	36C	P5	24W / 220W	535MiB / 8192MiB	32%	Default	N/A
<hr/>							
<hr/>							
Processes:							
GPU	GI	CI	PID	Type	Process name	GPU Memory	
ID	ID					Usage	
<hr/>							
0	N/A	N/A	1123	G		59MiB	
0	N/A	N/A	1813	G		228MiB	
0	N/A	N/A	1945	G		92MiB	
0	N/A	N/A	3281	G		141MiB	
<hr/>							

Figura B.1: Resultado del comando `nvidia-smi` dentro del contenedor.

A pesar de que el comando se ha ejecutado dentro de una imagen con CUDA 11.0, el resultado muestra CUDA 11.6 ya que accede directamente al *driver* del *host* y muestra la versión más alta de CUDA soportada por el *driver* instalado. Se puede encontrar más información sobre esto en: <https://github.com/NVIDIA/nvidia-docker/issues/1237#issuecomment-703524943>.

B.3.4. Construcción de la imagen de Docker

Para construir la imagen de Docker, se emplea el Dockerfile incluido en el repositorio, que parte de una imagen con CUDA 11.1 y PyTorch 1.9.0 e instala el *software* necesario para ejecutar los *scripts* correctamente. El comando a utilizar (si nos encontramos en la misma carpeta que el Dockerfile) es el siguiente, donde los argumentos `USER_ID` y `GROUP_ID` se utilizan para crear un usuario en la imagen que corresponda con el del *host* y evitar problemas de permisos, y `$tag` será el nombre que se le dará a la imagen:

```
$ docker build \
--build-arg USER_ID=$(id -u) --build-arg GROUP_ID=$(id -g) --tag $tag .
```

Al ejecutar el comando anterior se genera un *build context* del directorio en el que nos encontramos, así que en caso de hacerlo en una carpeta con datos o archivos que no se van a emplear durante la construcción de la imagen podemos redactar un archivo `.dockerignore` para no considerarlos.

B.3.5. Instalación de wandb (Opcional)

Para poder gestionar las búsquedas de hiperparámetros desde el *host* sin necesidad de entrar a un contenedor, es recomendable (pero no estrictamente necesario) instalar wandb, bien a través del instalador de paquetes de Python (pip) o creando un entorno de Anaconda, entre otras. Tener este paquete instalado, además, facilita el paso de credenciales a los contenedores de Docker a través del comando `wandb docker-run`

B.4. Ejecución

Para poder ejecutar los *scripts* de este proyecto, es necesario crear un contenedor de la imagen previamente construida. Para esto, empleamos el siguiente comando (desde el directorio TFM):

```
$ wandb docker-run --gpus all --rm -it --name ${tag}_container --ipc=host \
-p 8888:8888 -v $(pwd):/workspace -v /tmp/.X11-unix:/tmp/.X11-unix -e DISPLAY $tag
```

En este comando, `$tag` debe corresponderse con el nombre de la imagen creada anteriormente. El resto de opciones de este comando son: `--gpus all`, para usar en el contenedor todas las GPUs disponibles; `--rm`, para eliminar el contenedor y su sistema de ficheros al cerrarlo; `-it`, para hacerlo interactivo y abrir una sesión al iniciarla; `--ipc=host` para facilitar la comunicación entre procesos de PyTorch; `-p 8888:8888`, para mapear los puertos correspondientes y poder abrir cuadernos jupyter que se ejecutan dentro del contenedor a través del navegador del *host*; por último, la opción `-v $(pwd):/workspace` monta el directorio actual (TFM) dentro del contenedor, facilitando así el acceso al *dataset* y al código fuente que se encuentran en el *host*.

Además de las ya mencionadas, hay dos opciones más, `-v /tmp/.X11-unix:/tmp/.X11-unix` `-e DISPLAY`, empleadas para habilitar la visualización y uso de GUIs (por ejemplo, la función `imshow` de OpenCV) dentro del contenedor.

Tal y como se ha comentado previamente, si wandb no está instalado en el *host*, se puede sustituir `wandb docker-run` por `docker run`, siendo entonces necesario configurar las credenciales de wandb una vez iniciado el contenedor.

Una vez iniciado el contenedor, la ejecución de los *scripts* se puede agrupar en:

- **Gestionada por wandb:** Como es el caso de los *scripts* `attention_complexity.py`, `inference_speed.py` y `train.py`. Estos procesos iteran sobre distintas configuraciones de modelos y por lo tanto es necesario lanzarlos a través del gestor de ejecuciones de wandb, que se encarga de distribuir la ejecución de las configuraciones en tantos equipos como dispongamos.

1. Creamos el *sweep* en el *host* con el archivo YAML correspondiente.

```
$ wandb sweep <fichero-YAML>
```

2. Lanzamos agentes asociados a dicho *sweep* en los contenedores disponibles (tanto en local como en *cloud* (desde sus respectivos directorios DPT/)).

```
$ wandb agent <usuario>/<nombre-proyecto>/<sweep-id>
```

- **No gestionada por wandb:** Por otro lado, la inferencia y evaluación de un modelo concreto en el conjunto de *test* requiere que se especifique la configuración deseada y el fichero de pesos en el *script* `run_monodepth.py`. Es posible ejecutar este *script* de dos formas (también desde el directorio DPT/):

- Directamente, ejecutando `run_monodepth.py`. Por defecto, infiere la profundidad de las imágenes en la carpeta `input/` y almacena el resultado en la carpeta `output_monodepth/`. → `$ python run_monodepth.py`
- A través de `run_eval_with_pngs.sh`, que elimina los resultados anteriores, infiere con los parámetros adecuados el resultado en el conjunto de *test*, y calcula una serie de métricas con las anotaciones correspondientes utilizando el *script* proporcionado en [77] (`eval_with_pngs.py`). → `$./run_eval_with_pngs.sh`

B.5. Despliegue en la nube

Tal y como se ha mencionado en el [Apartado 4.1](#), antes de crear una imagen de la instancia que se replicará dentro de Google Cloud, es necesario cargar los conjuntos de datos y configurarla correctamente. En la página siguiente, se encuentra el *script* necesario para configurar la máquina inicial (`setup-gcp-gpu.sh`).

Antes de ejecutarlo, es necesario tener disponibles en un *bucket* de Google Cloud los datos que se quieren copiar en la máquina.

Una vez el *script* está en la máquina virtual, basta con ejecutarlo para que se copien los conjuntos de datos y el código del repositorio (siempre y cuando, evidentemente, se hayan cargado en el *bucket*). Después de copiar los datos, se procederá a instalar los controladores de NVIDIA, Docker, el NVIDIA Container Toolkit, Python y wandb.

Tras instalar correctamente todos los paquetes, se construye la imagen de Docker con `docker-build.sh`, que usa el Dockerfile del repositorio para definir la imagen de Docker.

docker-build.sh

```
[[ $# -eq 0 ]] && { echo "Usage: $0 tag"; exit 1; }
docker build --build-arg USER_ID=$(id -u) --build-arg GROUP_ID=$(id -g) --tag $1 .
```

Dockerfile

```
FROM pytorch/pytorch:1.9.0-cuda11.1-cudnn8-runtime
ARG USER_ID
ARG GROUP_ID
WORKDIR /workspace
RUN apt-get update ##[edited]
RUN apt-get install ffmpeg libsm6 libxext6 -y
RUN python3 -m pip install --upgrade pip
COPY DPT/requirements.txt requirements.txt
RUN pip3 install -r requirements.txt
RUN addgroup --gid $GROUP_ID user
RUN adduser --disabled-password --gecos '' --uid $USER_ID --gid $GROUP_ID user
USER user
ENTRYPOINT ["/bin/bash"]
```

Por último, cabe mencionar que para usar wandb dentro de las máquinas virtuales es necesario configurar la API KEY correctamente a través de la variable de entorno `WANDB_API_KEY`.

Una vez configurada la instancia, podemos crear una imagen de esta desde el panel de Google Cloud y configurar su ejecución para que cada vez que se ponga en marcha una instancia se ejecuten los siguientes dos comandos, el primero para poner en marcha un contenedor a partir de la imagen (de Docker) creada, y el segundo para ejecutar el agente de `wandb` que se quiera lanzar.

```
wandb docker-run -d --gpus all --rm -it --name pytorch_container --ipc=host \
-p 8888:8888 -v $(pwd):/workspace pytorch

docker exec pytorch_container sh -c "cd dpt && python -m wandb agent <agente>"
```

setup-gcp-gpu.sh

```
#!/bin/bash
# Run without sudo
# Clone dataset and code
gsutil -m cp -r gs://tfm-dpt/DPT .
gsutil -m cp -r gs://tfm-dpt/data .
gsutil cp -r gs://tfm-dpt/.dockerignore .
gsutil cp -r gs://tfm-dpt/Dockerfile .
gsutil cp -r gs://tfm-dpt/docker-build.sh .
# Install NVIDIA Drivers
sudo apt-get update
sudo apt-get upgrade -y
sudo apt-get install wget gcc make -y
sudo apt-get install linux-headers-`uname -r` -y
sudo wget \
https://us.download.nvidia.com/tesla/470.82.01/NVIDIA-Linux-x86_64-470.82.01.run
sudo sh NVIDIA-Linux-x86_64-470.82.01.run --silent
# Install Docker CE
sudo apt-get remove docker docker-engine docker.io containerd runc
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg lsb-release
curl -fsSL https://download.docker.com/linux/debian/gpg \
| sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) \
signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \
https://download.docker.com/linux/debian $(lsb_release -cs) stable" \
| sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io -y
# Post-installation
sudo groupadd docker
sudo usermod -aG docker $USER # -> $USER==root if the script is run with sudo
newgrp docker
# NVIDIA Container Toolkit
distribution=$( . /etc/os-release; echo $ID$VERSION_ID ) \
&& curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | sudo apt-key add - \
&& curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list \
| sudo tee /etc/apt/sources.list.d/nvidia-docker.list
sudo apt-get update
sudo apt-get install -y nvidia-docker2
sudo systemctl restart docker
# Pip and wandb
sudo apt update
sudo apt install python3-pip -y
pip3 install wandb
# Build docker image
chmod u+x docker-build.sh
./docker-build.sh pytorch
```

Apéndice C

Demostración de la independencia de la escala

Tanto la métrica SILog como la función de pérdida utilizada en este trabajo provienen de la ecuación presentada en el trabajo de Eigen et al. [75]. Esta ecuación, es invariante a la escala si $\lambda = 1$. En la siguiente demostración, se puede comprobar cómo incluir un factor de escala α que multiplica la profundidad obtenida por la red \hat{d} (segunda línea) no altera la ecuación.

$$\begin{aligned}
L(\hat{d}, d) &= \frac{1}{n} \sum_p (\ln \hat{d}_p - \ln d_p)^2 - \frac{1}{n^2} \left(\sum_p (\ln \hat{d}_p - \ln d_p) \right)^2 \\
&= \frac{1}{n} \sum_p (\ln \alpha \hat{d}_p - \ln d_p)^2 - \frac{1}{n^2} \left(\sum_p (\ln \alpha \hat{d}_p - \ln d_p) \right)^2 \\
&= \frac{1}{n} \sum_p (\ln \alpha + \ln \frac{\hat{d}_p}{d_p})^2 - \frac{1}{n^2} \left(\sum_p (\ln \alpha + \ln \frac{\hat{d}_p}{d_p}) \right)^2 \\
&= \frac{1}{n} \sum_p (\beta + \ln \frac{\hat{d}_p}{d_p})^2 - \frac{1}{n^2} \left(\sum_p (\beta + \ln \frac{\hat{d}_p}{d_p}) \right)^2 \\
&= \frac{1}{n} \sum_p (\beta^2 + \ln \frac{\hat{d}_p}{d_p})^2 + 2\beta \ln \frac{\hat{d}_p}{d_p} - \frac{1}{n^2} \left(\beta n + \sum_p (\ln \frac{\hat{d}_p}{d_p}) \right)^2 \\
&= \frac{1}{n} (\beta^2 n + \sum_p (\ln \frac{\hat{d}_p}{d_p})^2 + \sum_p (2\beta \ln \frac{\hat{d}_p}{d_p})) - \frac{1}{n^2} \left(\beta^2 n^2 + (\sum_p (\ln \frac{\hat{d}_p}{d_p}))^2 + 2\beta n \sum_p (\ln \frac{\hat{d}_p}{d_p}) \right) \\
&= \beta^2 + \frac{\sum_p (\ln \frac{\hat{d}_p}{d_p})^2}{n} + \frac{\sum_p (2\beta \ln \frac{\hat{d}_p}{d_p})}{n} - \left(\beta^2 + \frac{(\sum_p (\ln \frac{\hat{d}_p}{d_p}))^2}{n^2} + \frac{2\beta n \sum_p (\ln \frac{\hat{d}_p}{d_p})}{n^2} \right) \\
&= \beta^2 - \beta^2 + \frac{\sum_p (2\beta \ln \frac{\hat{d}_p}{d_p})}{n} - \frac{2\beta \sum_p (\ln \frac{\hat{d}_p}{d_p})}{n} + \frac{\sum_p (\ln \frac{\hat{d}_p}{d_p})^2}{n} - \frac{(\sum_p (\ln \frac{\hat{d}_p}{d_p}))^2}{n^2} \\
&\quad = \frac{\sum_p (\ln \frac{\hat{d}_p}{d_p})^2}{n} - \frac{(\sum_p (\ln \frac{\hat{d}_p}{d_p}))^2}{n^2} \\
&= \frac{1}{n} \sum_p (\ln \hat{d}_p - \ln d_p)^2 - \frac{1}{n^2} \left(\sum_p (\ln \hat{d}_p - \ln d_p) \right)^2
\end{aligned}$$

Apéndice D

Ficheros de resultados

Además del repositorio de código, se ha creado un repositorio con una serie de archivos producidos durante el desarrollo del proyecto. Este repositorio está disponible públicamente en: <https://zenodo.org/record/6574941>

Dentro del repositorio se encuentran una serie de ficheros:

- model_configurations.csv: Fichero de valores separados por comas que relaciona cada uno de los 37 modelos disponibles con su configuración respecto a las modificaciones introducidas en este trabajo. Contiene 37 entradas en total.
- training.csv: Ficheros de valores separados por comas que incluye la configuración y duración del entrenamiento de cada modelo, así como sus resultados en los conjuntos de entrenamiento y de validación. Contiene 36 entradas en total.
- inference_metrics.csv: Fichero de valores separados por comas que incluye las medidas de velocidad de inferencia, el número de GFLOPs y el tamaño para cada una de las configuraciones posibles reduciendo y sin reducir el tamaño de la entrada y activando y desactivando la precisión mixta. Contiene 144 entradas en total.
- dpt_hybrid-kitti-e7069aae_020.pt: Fichero con los parámetros entrenados por los autores de la publicación original en imágenes sin reducir y con la configuración de DPT-Hybrid sin modificaciones.
- dpt_hybrid_custom-kitti-<model_code>_020.pt: En total, 36 ficheros siguen este patrón, correspondiendo cada uno de ellos con las parámetros entrenados de las distintas configuraciones de modelos estudiadas durante el desarrollo de este Trabajo Fin de Máster.
- test_output_samples: Directorio con los resultados de los 37 modelos finales en una muestra aleatoria de 18 imágenes del conjunto de test.

Bibliografía

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.
- [2] K. M. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Q. Davis, A. Mohiuddin, L. Kaiser, D. B. Belanger, L. J. Colwell, and A. Weller, “Rethinking attention with performers,” in *International Conference on Learning Representations*, 2021.
- [3] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, “An image is worth 16x16 words: Transformers for image recognition at scale,” in *International Conference on Learning Representations*, 2021.
- [4] R. Ranftl, A. Bochkovskiy, and V. Koltun, “Vision transformers for dense prediction,” 2021.
- [5] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.
- [6] R. Hartley and A. Zisserman, *Two-View Geometry*, p. 237–238. Cambridge University Press, 2 ed., 2004.
- [7] M. Kalloniatis and C. Luu, *Webvision: The Organization of the Retina and Visual System*. 2005.
- [8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.
- [9] J. Redmon and A. Farhadi, “Yolov3: An incremental improvement,” 2018.
- [10] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015* (N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, eds.), (Cham), pp. 234–241, Springer International Publishing, 2015.
- [11] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

- [12] M. Tan and Q. Le, “EfficientNet: Rethinking model scaling for convolutional neural networks,” in *Proceedings of the 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, pp. 6105–6114, PMLR, 09–15 Jun 2019.
- [13] H. Jung, Y. Kim, D. Min, C. Oh, and K. Sohn, “Depth prediction from a single image with conditional adversarial networks,” in *2017 IEEE International Conference on Image Processing (ICIP)*, pp. 1717–1721, 2017.
- [14] K. Xian, J. Zhang, O. Wang, L. Mai, Z. Lin, and Z. Cao, “Structure-guided ranking loss for single image depth prediction,” in *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.
- [15] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, pp. 436–444, May 2015.
- [16] Y. Bengio, A. C. Courville, and P. Vincent, “Unsupervised feature learning and deep learning: A review and new perspectives,” *CoRR, abs/1206.5538*, vol. 1, p. 2012, 2012.
- [17] Y. Ouali, C. Hudelot, and M. Tami, “An overview of deep semi-supervised learning,” 2020.
- [18] T. T. Nguyen, N. D. Nguyen, and S. Nahavandi, “Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications,” *IEEE Transactions on Cybernetics*, vol. 50, no. 9, pp. 3826–3839, 2020.
- [19] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain.,” *Psychological review*, vol. 65, no. 6, p. 386, 1958.
- [20] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.
- [21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning Representations by Back-propagating Errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.
- [22] S. Raschka, “Model evaluation, model selection, and algorithm selection in machine learning,” *CoRR, vol. abs/1811.12808*, 2018.
- [23] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th International Conference on International Conference on Machine Learning, ICML’10*, (Madison, WI, USA), p. 807–814, Omnipress, 2010.
- [24] D. Hendrycks and K. Gimpel, “Gaussian error linear units (gelus),” *arXiv preprint arXiv:1606.08415*, 2016.
- [25] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient backprop,” in *Neural networks: Tricks of the trade*, pp. 9–48, Springer, 2012.
- [26] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” *Journal of Machine Learning Research*, vol. 12, no. 61, pp. 2121–2159, 2011.
- [27] G. Hinton, “Overview of mini-batch gradient descent.”
- [28] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *ICLR (Poster)*, 2015.
- [29] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” in *International Conference on Learning Representations*, 2019.

- [30] N. Keskar, J. Nocedal, P. Tang, D. Mudigere, and M. Smelyanskiy, “On large-batch training for deep learning: Generalization gap and sharp minima,” 2017. 5th International Conference on Learning Representations, ICLR 2017 ; Conference date: 24-04-2017 Through 26-04-2017.
- [31] M. Minsky and S. Papert, “Perceptrons.,” 1969.
- [32] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder–decoder for statistical machine translation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (Doha, Qatar), pp. 1724–1734, Association for Computational Linguistics, Oct. 2014.
- [33] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [34] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, *Object Recognition with Gradient-Based Learning*, pp. 319–345. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999.
- [35] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” in *European conference on computer vision*, pp. 818–833, Springer, 2014.
- [36] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.
- [37] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, “Mnasnet: Platform-aware neural architecture search for mobile,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2815–2823, 2019.
- [38] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520, 2018.
- [39] J. Hu, L. Shen, and G. Sun, “Squeeze-and-excitation networks,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 7132–7141, 2018.
- [40] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” Jan. 2015. 3rd International Conference on Learning Representations, ICLR 2015 ; Conference date: 07-05-2015 Through 09-05-2015.
- [41] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, “Show, attend and tell: Neural image caption generation with visual attention,” in *Proceedings of the 32nd International Conference on Machine Learning* (F. Bach and D. Blei, eds.), vol. 37 of *Proceedings of Machine Learning Research*, (Lille, France), pp. 2048–2057, PMLR, 07–09 Jul 2015.
- [42] T. Xiao, Y. Xu, K. Yang, J. Zhang, Y. Peng, and Z. Zhang, “The application of two-level attention models in deep convolutional neural network for fine-grained image classification,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 842–850, 2015.
- [43] C. Cao, X. Liu, Y. Yang, Y. Yu, J. Wang, Z. Wang, Y. Huang, L. Wang, C. Huang, W. Xu, D. Ramanan, and T. S. Huang, “Look and think twice: Capturing top-down visual attention with feedback convolutional neural networks,” in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2956–2964, 2015.

- [44] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, “Efficient Transformers: A Survey,” *arXiv e-prints*, p. arXiv:2009.06732, Sept. 2020.
- [45] J. Qiu, H. Ma, O. Levy, W.-t. Yih, S. Wang, and J. Tang, “Blockwise self-attention for long document understanding,” in *Findings of the Association for Computational Linguistics: EMNLP 2020*, (Online), pp. 2555–2565, Association for Computational Linguistics, Nov. 2020.
- [46] P. Ramachandran, N. Parmar, A. Vaswani, I. Bello, A. Levskaya, and J. Shlens, “Stand-alone self-attention in vision models,” in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.
- [47] R. Child, S. Gray, A. Radford, and I. Sutskever, “Generating Long Sequences with Sparse Transformers,” *arXiv e-prints*, p. arXiv:1904.10509, Apr. 2019.
- [48] I. Beltagy, M. E. Peters, and A. Cohan, “Longformer: The long-document transformer,” 2020.
- [49] P. J. Liu*, M. Saleh*, E. Pot, B. Goodrich, R. Sepassi, L. Kaiser, and N. Shazeer, “Generating wikipedia by summarizing long sequences,” in *International Conference on Learning Representations*, 2018.
- [50] J. Ho, N. Kalchbrenner, D. Weissenborn, and T. Salimans, “Axial attention in multidimensional transformers,” 2019.
- [51] N. Kitaev, L. Kaiser, and A. Levskaya, “Reformer: The efficient transformer,” in *International Conference on Learning Representations*, 2020.
- [52] A. Roy, M. Saffar, A. Vaswani, and D. Grangier, “Efficient Content-Based Sparse Attention with Routing Transformers,” *Transactions of the Association for Computational Linguistics*, vol. 9, pp. 53–68, 02 2021.
- [53] Q. Zhang and Y. Yang, “Rest: An efficient transformer for visual recognition,” *arXiv preprint arXiv:2105.13677v2*, 2021.
- [54] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, “Linenformer: Self-attention with linear complexity,” 2020.
- [55] N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, N. Shazeer, A. Ku, and D. Tran, “Image transformer,” in *Proceedings of the 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 4055–4064, PMLR, 10–15 Jul 2018.
- [56] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, “End-to-end object detection with transformers,” in *Computer Vision – ECCV 2020* (A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, eds.), (Cham), pp. 213–229, Springer International Publishing, 2020.
- [57] S. F. Bhat, I. Alhashim, and P. Wonka, “Adabins: Depth estimation using adaptive bins,” 2020.
- [58] J. Chen, Y. Lu, Q. Yu, X. Luo, E. Adeli, Y. Wang, L. Lu, A. L. Yuille, and Y. Zhou, “Transunet: Transformers make strong encoders for medical image segmentation,” *arXiv preprint arXiv:2102.04306*, 2021.

- [59] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, “Swin transformer: Hierarchical vision transformer using shifted windows,” *arXiv preprint arXiv:2103.14030*, 2021.
- [60] C. Zhao, Q. Sun, C. Zhang, Y. Tang, and F. Qian, “Monocular depth estimation based on deep learning: An overview,” *Science China Technological Sciences*, vol. 63, p. 1612–1627, Jun 2020.
- [61] T. Zhou, M. Brown, N. Snavely, and D. G. Lowe, “Unsupervised learning of depth and ego-motion from video,” in *CVPR*, 2017.
- [62] S. Vijayanarasimhan, S. Ricco, C. Schmid, R. Sukthankar, and K. Fragkiadaki, “Sfm-net: Learning of structure and motion from video,” 2017.
- [63] J. Bian, Z. Li, N. Wang, H. Zhan, C. Shen, M.-M. Cheng, and I. Reid, “Unsupervised scale-consistent depth and ego-motion learning from monocular video,” in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.
- [64] C. Godard, O. M. Aodha, M. Firman, and G. Brostow, “Digging into self-supervised monocular depth estimation,” in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 3827–3837, 2019.
- [65] C. Wang, J. M. Buenaposada, R. Zhu, and S. Lucey, “Learning depth from monocular videos using direct methods,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2022–2030, 2018.
- [66] Z. Yin and J. Shi, “Geonet: Unsupervised learning of dense depth, optical flow and camera pose,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1983–1992, 2018.
- [67] F. Tosi, F. Aleotti, M. Poggi, and S. Mattoccia, “Learning monocular depth estimation infusing traditional stereo knowledge,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9791–9801, 2019.
- [68] Y. Luo, J. Ren, M. Lin, J. Pang, W. Sun, H. Li, and L. Lin, “Single view stereo matching,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 155–163, 2018.
- [69] N. Smolyanskiy, A. Kamenev, and S. Birchfield, “On the importance of stereo for accurate depth estimation: An efficient semi-supervised deep neural network approach,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 1120–11208, 2018.
- [70] J. Xie, R. B. Girshick, and A. Farhadi, “Deep3d: Fully automatic 2d-to-3d video conversion with deep convolutional neural networks,” in *ECCV*, 2016.
- [71] Y. Xu, X. Zhu, J. Shi, G. Zhang, H. Bao, and H. Li, “Depth completion from sparse lidar data with depth-normal constraints,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.
- [72] F. Ma, G. V. Cavalheiro, and S. Karaman, “Self-supervised sparse-to-dense: Self-supervised depth completion from lidar and monocular camera,” 2019.
- [73] M. Hu, S. Wang, B. Li, S. Ning, L. Fan, and X. Gong, “Towards precise and efficient image guided depth completion,” 2021.

- [74] L. He, C. Chen, T. Zhang, H. Zhu, and S. Wan, “Wearable depth camera: Monocular depth estimation via sparse optimization under weak supervision,” *IEEE Access*, vol. 6, pp. 41337–41345, 2018.
- [75] D. Eigen, C. Puhrsch, and R. Fergus, “Depth map prediction from a single image using a multi-scale deep network,” in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS’14, (Cambridge, MA, USA), p. 2366–2374, MIT Press, 2014.
- [76] D. Eigen and R. Fergus, “Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture,” in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2650–2658, 2015.
- [77] J. H. Lee, M.-K. Han, D. W. Ko, and I. H. Suh, “From big to small: Multi-scale local planar guidance for monocular depth estimation,” *arXiv preprint arXiv:1907.10326*, 2019.
- [78] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in *Advances in Neural Information Processing Systems* (Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, eds.), vol. 27, Curran Associates, Inc., 2014.
- [79] D. Kim, W. Ga, P. Ahn, D. Joo, S. Chun, and J. Kim, “Global-local path networks for monocular depth estimation with vertical cutdepth,” *arXiv preprint arXiv:2201.07436*, 2022.
- [80] G. Lin, A. Milan, C. Shen, and I. Reid, “Refinenet: Multi-path refinement networks for high-resolution semantic segmentation,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.
- [81] R. Ranftl, K. Lasinger, D. Hafner, K. Schindler, and V. Koltun, “Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2020.
- [82] Z. Li and N. Snavely, “Megadepth: Learning single-view depth prediction from internet photos,” in *Computer Vision and Pattern Recognition (CVPR)*, 2018.
- [83] S. Vadera and S. Ameen, “Methods for pruning deep neural networks,” 2020.
- [84] S. Han, J. Pool, J. Tran, and W. Dally, “Learning both weights and connections for efficient neural network,” in *Advances in Neural Information Processing Systems* (C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, eds.), vol. 28, Curran Associates, Inc., 2015.
- [85] A. Polyak and L. Wolf, “Channel-level acceleration of deep face representations,” *IEEE Access*, vol. 3, pp. 2163–2175, 2015.
- [86] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, “Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures,” *arXiv e-prints*, p. arXiv:1607.03250, July 2016.
- [87] Y. LeCun, J. Denker, and S. Solla, “Optimal brain damage,” in *Advances in Neural Information Processing Systems* (D. Touretzky, ed.), vol. 2, Morgan-Kaufmann, 1990.
- [88] B. Hassibi, D. Stork, and G. Wolff, “Optimal brain surgeon: Extensions and performance comparisons,” in *Advances in Neural Information Processing Systems* (J. Cowan, G. Tesauro, and J. Alspector, eds.), vol. 6, Morgan-Kaufmann, 1994.

- [89] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, “Pruning convolutional neural networks for resource efficient inference,” 2017.
- [90] C. Wang, R. Grosse, S. Fidler, and G. Zhang, “EigenDamage: Structured pruning in the Kronecker-factored eigenbasis,” in *Proceedings of the 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, pp. 6566–6575, PMLR, 09–15 Jun 2019.
- [91] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, “Mixed precision training,” 2018.
- [92] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, pp. 357–362, Sept. 2020.
- [93] G. Bradski, “The OpenCV Library,” *Dr. Dobb’s Journal of Software Tools*, 2000.
- [94] FAIR, “fvcore.” <https://github.com/facebookresearch/fvcore>, 2021.
- [95] J. D. Hunter, “Matplotlib: A 2d graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.
- [96] M. L. Waskom, “seaborn: statistical data visualization,” *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021.
- [97] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.
- [98] R. Wightman, “Pytorch image models.” <https://github.com/rwightman/pytorch-image-models>, 2021.
- [99] P. Wang, “Performer pytorch.” <https://github.com/lucidrains/performer-pytorch>, 2021.
- [100] L. Biewald, “Experiment tracking with weights and biases,” 2020. Software available from wandb.com.
- [101] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A Large-Scale Hierarchical Image Database,” in *CVPR09*, 2009.
- [102] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, “Vision meets robotics: The kitti dataset,” *International Journal of Robotics Research (IJRR)*, 2013.
- [103] A. Geiger, P. Lenz, and R. Urtasun, “Are we ready for autonomous driving? the kitti vision benchmark suite,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.
- [104] J. Fritsch, T. Kuehnl, and A. Geiger, “A new performance measure and evaluation benchmark for road detection algorithms,” in *International Conference on Intelligent Transportation Systems (ITSC)*, 2013.

- [105] M. Menze and A. Geiger, “Object scene flow for autonomous vehicles,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.
- [106] C. Fellbaum, ed., *WordNet: An Electronic Lexical Database*. Language, Speech, and Communication, Cambridge, MA: MIT Press, 1998.
- [107] Y. Kim, H. Jung, D. Min, and K. Sohn, “Deep monocular depth estimation via integration of global and local predictions,” *IEEE Transactions on Image Processing*, vol. 27, no. 8, pp. 4131–4144, 2018.
- [108] K. Xian, C. Shen, Z. Cao, H. Lu, Y. Xiao, R. Li, and Z. Luo, “Monocular relative depth perception with web stereo data supervision,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.
- [109] C. Wang, S. Lucey, F. Perazzi, and O. Wang, “Web stereo video supervision for depth prediction from dynamic scenes,” 2019.
- [110] W. Wang, D. Zhu, X. Wang, Y. Hu, Y. Qiu, C. Wang, Y. Hu, A. Kapoor, and S. Scherer, “Tartanair: A dataset to push the limits of visual slam,” 2020.
- [111] P. Wang, X. Huang, X. Cheng, D. Zhou, Q. Geng, and R. Yang, “The apolloscape open dataset for autonomous driving and its application,” *IEEE transactions on pattern analysis and machine intelligence*, 2019.
- [112] Y. Yao, Z. Luo, S. Li, J. Zhang, Y. Ren, L. Zhou, T. Fang, and L. Quan, “Blendedmvs: A large-scale dataset for generalized multi-view stereo networks,” in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1787–1796, 2020.
- [113] Q. Wang, S. Zheng, Q. Yan, F. Deng, K. Zhao, and X. Chu, “Irs: A large naturalistic indoor robotics stereo dataset to train deep models for disparity and surface normal estimation,” in *2021 IEEE International Conference on Multimedia and Expo (ICME)*, (Los Alamitos, CA, USA), pp. 1–6, IEEE Computer Society, jul 2021.
- [114] W. Chen, Z. Fu, D. Yang, and J. Deng, “Single-image depth perception in the wild,” in *Advances in Neural Information Processing Systems* (D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, eds.), vol. 29, Curran Associates, Inc., 2016.
- [115] T. Schöps, J. L. Schönberger, S. Galliani, T. Sattler, K. Schindler, M. Pollefeys, and A. Geiger, “A multi-view stereo benchmark with high-resolution images and multi-camera videos,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.
- [116] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black, “A naturalistic open source movie for optical flow evaluation,” in *European Conf. on Computer Vision (ECCV)* (A. Fitzgibbon et al. (Eds.), ed.), Part IV, LNCS 7577, pp. 611–625, Springer-Verlag, Oct. 2012.
- [117] P. K. Nathan Silberman, Derek Hoiem and R. Fergus, “Indoor segmentation and support inference from rgbd images,” in *ECCV*, 2012.
- [118] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, “A benchmark for the evaluation of rgbd slam systems,” in *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.
- [119] H. Fu, M. Gong, C. Wang, K. Batmanghelich, and D. Tao, “Deep Ordinal Regression Network for Monocular Depth Estimation,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

- [120] T. Koch, L. Liebel, F. Fraundorfer, and M. Körner, “Evaluation of cnn-based single-image depth estimation methods,” in *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, September 2018.
- [121] C. Cadena, Y. Latif, and I. D. Reid, “Measuring the performance of single image depth estimation methods,” in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4150–4157, 2016.
- [122] M. Harris, “How to optimize data transfers in cuda c/c++,” Dec 2012.
- [123] P. Michel, O. Levy, and G. Neubig, “Are sixteen heads really better than one?,” in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.