



Memoria del trabajo realizado durante la estancia en el Grupo
de Sistemas Inteligentes

Estimación de Profundidad Monocular Online con Transformers Eficientes

Contexto y estado del arte

Guillermo Sánchez Brizuela

Universidad de Valladolid, curso 2020-2021

Índice

1. Introducción	3
1.1. Motivación	4
1.2. Objetivos	5
1.3. Estructura del documento	5
2. Marco teórico y Estado del Arte	6
2.1. Inteligencia Artificial, Aprendizaje automático, y Aprendizaje Profundo	6
2.2. Redes neuronales	6
2.2.1. Funciones de activación	7
2.2.2. Funciones de pérdida	7
2.2.3. Optimizadores	7
2.3. MLP	7
2.4. CNN	7
2.4.1. ResNet	8
2.4.2. Efficientnet	8
2.4.3. Fusionnet	8
2.5. RNN	8
2.6. Transformers	8
2.6.1. Arquitectura	8
2.6.2. Mecanismos de atención	9
2.6.3. Atención eficiente	10
2.6.4. Patrones fijos - Fixed Patterns (FP)	10
2.6.5. Aprendizaje de patrones - Learnable Patterns (LP)	11
2.6.6. Disminución de rango	11
2.6.7. Kernels	11
2.6.8. Performer	11
2.7. Transformers para visión artificial - ViT	11
2.8. Estimación de profundidades	12
2.8.1. Técnicas de estimación de profundidades	12
2.8.2. Aprendizaje automático no supervisado	13
2.8.3. Aprendizaje automático semisupervisado	13
2.8.4. Aprendizaje automático supervisado	14
2.8.5. DPT	14
2.9. Optimización de modelos	16
2.10. Redes neuronales - Transformers	19
2.10.1. Transformers para visión	19
2.11. Estimación de profundidades	19
2.12. Técnicas de mejora de eficiencia	19
2.13. Estimación de profundidades eficiente	19
3. Material y métodos	20
3.1. <i>Software y hardware</i> empleado	20
3.1.1. Software	20
3.1.2. Hardware	23
3.2. Cloud	23
3.3. Datasets	25
3.4. ImageNet	25
3.5. MIX6	25
3.5.1. KITTI	26
3.5.1.1. Datos disponibles	26

3.6. Arquitectura y capas	30
3.7. Warmstart	30
3.8. Función de pérdida	30
3.9. Data augmentation	30
3.10. Evaluación	30
3.10.1. Métricas	31
3.10.1.1. <i>Accuracy under a threshold</i>	31
3.10.1.2. <i>Mean Absolute Value of the Relative Error (Abs Rel)</i>	31
3.10.1.3. <i>Mean Squared Relative Error (Sq Rel)</i>	31
3.10.1.4. <i>Linear Root Mean Squared Error (RMSE)</i>	31
3.10.1.5. <i>Logarithmic Root Mean Squared Error (RMSElog)</i>	32
3.10.1.6. <i>Scale Invariant Logarithmic Error (SIlog)</i>	32
3.10.1.7. <i>Mean Logarithmic Error (Log10)</i>	32
3.10.1.8. Velocidad de procesamiento	33
3.11. Portabilidad (?) de los modelos	33
4. Modificaciones de la arquitectura y desarrollo	34
4.1. Entrenamiento	34
4.2. Reducción de tamaño de la entrada	34
4.3. Número de cabezas	34
4.4. Capas de atención eficiente	34
4.5. Cambio en los hooks del transformer y eliminación de las capas de atención posteriores	35
4.6. Cambio del backbone convolucional	37
5. Resultados	39
5.1. Resultados cuantitativos	39
5.1.1. Reducción de tamaño de la entrada	39
5.1.2. Número de cabezas	40
5.1.3. Capas de atención eficiente	40
5.1.4. Cambio en los hooks del transformer y eliminación de las capas de atención posteriores	41
5.1.5. Cambio del backbone convolucional	41
5.2. Resultados cualitativos	42
6. Discusión	43
7. Conclusiones y líneas futuras	44
Bibliografía	45

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). Es decir, existen infinitas escenas tridimensionales con la misma proyección perspectiva.

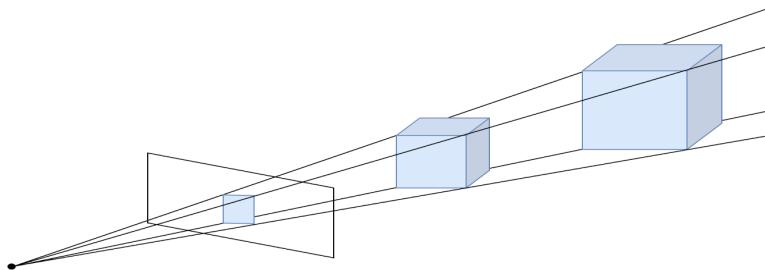


Figura 1: Proyección perspectiva y ambigüedad de escala.

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[1]. 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 esta) 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 oclusión en aplicaciones de realidad aumentada; generación de modelos 3D a través de fotografías (fotogrametría); efectos fotográficos en aplicaciones móviles (efecto *Bokeh*); etc.
- 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 sesgos cognitivos que los humanos hemos aprendido 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) [2] son: el tamaño relativo con el que observamos

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

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, si no 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 diseñados para trabajar con imágenes. 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. No obstante, en los últimos años han surgido otro tipo de arquitecturas, los *transformers* [3], que presentan resultados muy competitivos. 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 [5], 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.

Checkear
esto

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 que se plantearon al comenzarlo; a continuación, en el segundo capítulo se contextualiza el trabajo repasando las bases teóricas sobre las que se apoya su desarrollo, revisando también el Estado del Arte de estos campos; en el tercer capítulo, se presentan y justifican tanto los materiales empleados para el desarrollo del trabajo como 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. Por último, se incluye en el capítulo seis 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.

Poner algo de los anexos?

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, engloba multitud de disciplinas y técnicas con las que se TODO

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

2.2. Redes neuronales

Las redes neuronales (en inglés, *Neural Networks* - NN), son sistemas computacionales inspirados en una versión simplificada de las neuronas biológicas. Estas neuronas artificiales, están definidas por una serie de elementos: entradas, **pesos**, **bias**, **función de activación** y salida (Figura ??). El valor que toma la salida, viene definido por la Ecuación ??.

Uno de los puntos más importantes de las NN, es que con un conjunto de neuronas lo suficientemente grande y funciones de activación no lineales, las NN pueden aproximar cualquier función continua (*Universal approximation theorem* []). No obstante, 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, 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** []), que necesita: un **optimizador** (TODO decir que se explica 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 conocida). 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).

Calcular las derivadas parciales del error para todos los ejemplos sin actualizar los parámetros puede ser muy poco eficiente cuando el número de datos es grande, que suele ser lo normal. Por esta razón, el conjunto de datos que se consideran para llevar a cabo una actualización de los parámetros (lote o *batch*), suele ser un subconjunto muy reducido en comparación con el conjunto de datos disponible total. El tamaño de este subconjunto (**batch size**), influye en múltiples factores [?] del entrenamiento, como pueden ser la velocidad de entrenamiento, la velocidad de convergencia, o el mínimo al que se converge durante el entrenamiento.

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. En el trabajo de ... [?] distintas formas de hacer estas particiones. 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 esta introducción, se han mencionado algunos conceptos en los que se profundiza a continuación prestando especial atención a los elementos que aparecen en este trabajo.

2.2.1. Funciones de activación

Tal y como se puede ver en la Ecuación ??, 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 ??, es posible observar varias de estas funciones junto con sus ecuaciones.

Figura (lineal, sigmoide, tanh, relu, gelu

La primera de estas funciones de activación ?? es la función de activación lineal, esta función, no suele emplearse, ya que si concatenamos múltiples neuronas con activaciones lineales, serían equivalentes a una sola neurona con la función lineal equivalente. Por esta razón, son funciones como la sigmoide ??, la tangente hiperbólica ??, ReLU [] (*Rectified Linear Unit*) ?? o GELU [] (*Gaussian Error Linear Unit*) ??.

TODO: Hablar de cada una un poco por encima.

<https://towardsdatascience.com/if-rectified-linear-units-are-linear-how-do-they-add-nonlinearities-10f3a2a2a2>
- Mencionar que las ReLU son lineales a medias.

2.2.2. Funciones de pérdida

Lorem ipsum

2.2.3. Optimizadores

Los optimizadores son algoritmos que a partir de los gradientes de los parámetros respecto de una función de perdida (también llamada función objetivo), ajustan estos parámetros para minimizar dicha perdida. Existen distintos optimizadores, algunos de los más empleados: SGD, Adagrad, RMSProp (primero el que tenga más sentido), Adam, Adamw

Our method is designed to combine the advantages of two recently popular methods: AdaGrad (Duchi et al., 2011), which works well with sparse gradients, and RMSProp (Tieleman & Hinton, 2012), which works well in on-line and non-stationary settings;

2.3. MLP

Lorem

2.4. 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 crecería enormemente a la hora de trabajar con imágenes cuyo tamaño no fuese especialmente reducido. Las redes convolucionales, no solo lidian con este problema, si no que 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 [1]. Estos *kernels* son pequeñas matrices 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. Los *kernels*, normalmente tienden a extraer características de más alto nivel cuanto más adelante están en la red. Esto significa que las características que se extraen en las capas iniciales son características de muy bajo nivel (líneas verticales, horizontales, curvas, esquinas, etc.), mientras que las de las últimas capas extraen características más complejas.

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

- Pooling: TODO
- Deconvolución / Convención transpuesta: TODO

2.4.1. ResNet

Lorem ipsum

2.4.2. Efficientnet

Lorem ipsum

2.4.3. Fusionnet

Lorem ipsum

2.5. RNN

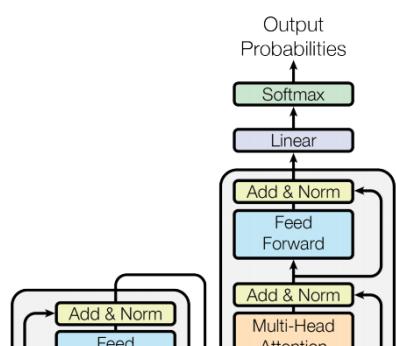
Lorem

2.6. Transformers

La arquitectura inicial del *transformer* (Figura 2), propuesta en *Attention is All You Need* [3], 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) - 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) - otra frase -. Una de las principales diferencias frente a los modelos *encoder-decoder* basados en redes recurrentes, es que a pesar de que el modelo sigue siendo auto-regresivo, es decir, sigue utilizando los elementos generados por la salida del modelo como entrada para el siguiente elemento a generar, en este caso la secuencia de entrada no está alineada temporalmente con la ejecución del modelo y por lo tanto puede paralelizarse todo el procesamiento de dicha secuencia, acelerando entrenamiento e inferencia.

2.6.1. Arquitectura

En el **encoder** (parte izquierda de la Figura 2), 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*² 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), 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 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 [35], buscan simular la atención cognitiva y han sido previamente empleados en redes recurrentes [36] y convolucionales [37, 38] para aprender qué partes de la entrada son más relevantes en la tarea a completar. Sin embargo, en [3], 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 [35] y 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*, elementos principales de los *transformers*, que aparecen de dos formas distintas:

- 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*.

²Operación 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

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 3) 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.

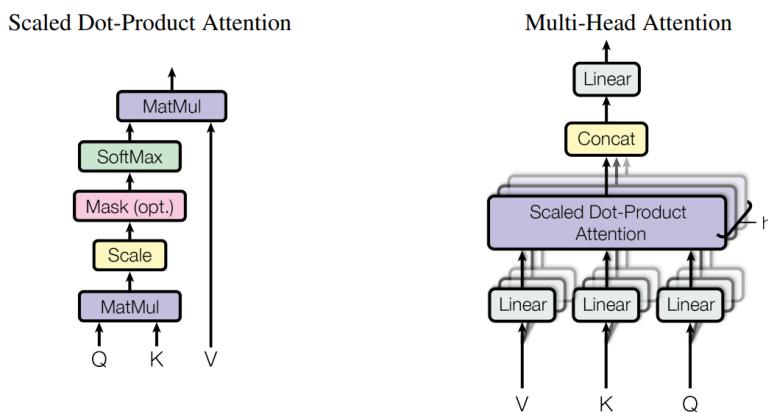


Figura 3: Producto escalar para el cálculo de la atención (izquierda) y bloque de *Multi-Head Attention* (derecha). Fuente: [3]

Estos mecanismos de atención, sin embargo, son 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 esto 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 [79], estas modificaciones de las capas de atención pueden agruparse en:

Hablar del performer en más detalle

2.6.4. 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* [80] y *Local Attention* [81]; accediendo a la secuencia en intervalos previamente definidos, como en *Sparse Transformer* [82] o *Longformer* [83] 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*** [84].

2.6.5. 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*** [85], 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*** [86] 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*** [87], que está enfocado a imágenes y emplea convoluciones separables para reducir las dimensiones de las entradas al mecanismo de atención.

2.6.6. 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***, [88] que presenta una complejidad $O(n)$

2.6.7. Kernels

Por último, y aunque podrían entrar dentro del grupo de disminución de rango, existen enfoques que emplean kernelización en los mecanismos de atención para evitar el cálculo explícito de la matriz $n \times n$. Un ejemplo de este tipo de enfoques es el propuesto en [89], que de nuevo reduce la complejidad a $O(n)$.

2.6.8. Performer

Este trabajo se centra en el mecanismo propuesto por .., el performer, para...

2.7. Transformers para visión artificial - ViT

A la hora de aplicar la arquitectura de los *transformers* a problemas de tratamiento de imágenes, surge un problema importante. La complejidad del mecanismo 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* [39]) o aplicar convoluciones para reducir el tamaño de la secuencia de entrada [40].

Sin embargo, la solución propuesta en *An Image is Worth 16x16 Words* con el *Vision Transformer* [41] 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 [5, 32, 42, 43]. 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 4). Como nota, esta arquitectura, inicialmente propuesta para realizar clasificación, solamente emplea el *encoder* del *transformer*.



Figura 4: Separación de una imagen en fragmentos siguiendo la propuesta de [41]. Fuente: Elaboración propia inspirada en [41]

Hablar del vit hybrid y mencionar que es el que se emplea

2.8. Estimación de profundidades

Repasar estos tres parrafo

La estimación de profundidades, tanto cuando es llevada a cabo por humanos como por máquinas, consiste en detectar la distancia relativa entre todo aquello que se ve. Para conseguir esto, nuestro cerebro se apoya principalmente en la disparidad existente entre las imágenes que capturan cada uno de nuestros ojos, 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. Esto se conoce como estereovisión.

Si nos tapamos un ojo, deberíamos perder esta capacidad para percibir la profundidad, sin embargo, esto no ocurre debido a que el cerebro explota lo que se conocen como pistas monoculares, que aportan información sobre la distancia a la que están las cosas a partir de las oclusiones, sombras, perspectiva, el tamaño esperado de un objeto, el paralelo, etc.

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. Para este tipo de estimación (monocular), entran en juego los modelos de aprendizaje automático, que han demostrado una gran capacidad para explotar este tipo de construcciones en todo tipo de tareas, no siendo la estimación de profundidades diferente. No obstante, a continuación se enumeran los distintos enfoques existentes, tanto tradicionales como basados en aprendizaje automático, en los cuales se profundizará especialmente.

2.8.1. Técnicas de estimación de profundidades

La estimación de profundidades se ha intentado resolver de múltiples maneras [44]. 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 fijas entre sí y conocidas. Estos métodos, sin embargo, basan su funcionamiento en el emparejamiento de puntos claves (*feature points*) que deben

encontrarse en ambas imágenes, y por lo tanto necesitan 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 LIDAR³), 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 (por múltiples factores, 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 datos. Normalmente, estas soluciones emplean secuencias (vídeos) de imágenes monoculares, extrayendo automáticamente a partir de estas una señal supervisora con distintos métodos. En [45] 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 [45, 46] como en técnicas geométricas [47, 29], para lidiar con los objetos dinámicos. Otros enfoques, con la misma idea subyacente, sustituyen la red de estimación de posición con métodos de odometría visual⁴ tradicional [48], 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 [46, 49].

2.8.3. Aprendizaje automático semisupervisado

Debido 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 [50, 51].
- Generación de los mapas de disparidad entre parejas de imágenes para estereovisión [52, 53], donde la señal supervisora es la disparidad obtenida mediante técnicas tradicionales de

³Por 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.

⁴La 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.

estereovisión.

- Utilización de datos etiquetados de forma dispersa (frecuentemente obtenidos con dispositivos LIDAR), que o bien emplean imágenes no etiquetadas para densificar dichas representaciones [54, 55, 56] o añaden la información del LIDAR en la función de perdidas para después inferir a partir de imagen monocular [57].

2.8.4. Aprendizaje automático supervisado

Pese a la dificultad para obtener datos etiquetados correctamente, los enfoques supervisados siguen siendo los que mejores resultados ofrecen y por lo tanto han sido y siguen siendo extensamente estudiados. El primero de estos modelos fue propuesto en [28], 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 [58] a través del cálculo de los gradientes de la diferencia entre el resultado y el objetivo. Otra función de perdidas popular es la perdida *Berhu* [59]. Además de aquellos basados en arquitecturas *encoder-decoder*, también hay propuestas con arquitecturas de aprendizaje adversario [60] donde el discriminador trata de distinguir entre la profundidad generada y la real [61].

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. Entre estos modelos, destacan:

- **Dense Prediction Transformers:** Propuesto en [5], emplea como *backbones* los distintos modelos propuestos en [41], de donde se extraen representaciones intermedias de diferentes resoluciones que posteriormente se fusionan empleando capas convolucionales. Esta red, cuenta con dos cabezas, una pre-entrenada para estimación de imágenes monocular y otra para segmentación semántica. Este modelo, proporciona distancias relativas, es decir, no aporta información métrica.
- **AdaBins:** Presentado en [32], también emplea como base los *vision transformers* propuestos en [41]. A diferencia del modelo anterior, *AdaBins* (*Adaptative Bins*) propone un bloque adicional que 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 más suavizada. A diferencia del anterior, aunque con ligeramente peores resultados, *AdaBins* sí que proporciona distancias en metros.

Los tiempos de inferencia de estos modelos han sido evaluados en distintos dispositivos y se presentan en la sección ??.

2.8.5. DPT

Introducción

Añadir citas aquí

Las arquitecturas de estimación de profundidades se basan en redes convolucionales, normalmente de tipo *encoder-decoder*. Las líneas de investigación actuales se centran en el *decoder* y sus estrategias de agregación de información, sin embargo, DPT 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.

Esto puede que haya que moverlo a otra zona del documento para argumentar por qué se usan transformers y no redes convolucionales

En el artículo, argumentan que los *backbones* convolucionales reducen dimensionalmente la imagen de forma progresiva para extraer sus mapas de características con distintos niveles de abstracción mientras se mantienen razonables los requisitos computacionales y de memoria. Este tipo de operaciones, que han ofrecido muy buenos resultados en todo tipo de tareas de visión artificial, presentan ciertos inconvenientes críticos en la estimación de profundidades, principalmente la resolución y granularidad de los mapas de características extraídos. Esto, es un problema debido a que para estimar la profundidad de una imagen con la mayor exactitud posible, sería conveniente que los mapas de características extraídos de la imagen mantuvieran su resolución original (o cercana a esta).

Pese a que se han planteado distintos métodos para reducir la perdida de resolución en los mapas de características (emplear imágenes más grandes, convoluciones dilatadas, skip connections, conexión de representaciones internamente, etc.)

citar todo esto si se queda

, esta perdida de granularidad es inherente a la operación de convolución tál y como se aplica en este tipo de redes. Para solucionar esto, DPT emplea transformers (en concreto, vision transformers) como *backbone* para evitar una reducción explícita de la resolución de las características extraídas de la imagen y ampliar el campo receptivo con el que opera la red (es posible usar información de toda la imagen en cualquier etapa de la arquitectura - campo receptivo global -, mientras que las capas convolucionales tienen campos receptivos locales mucho menores).

El trabajo previo repartirlo por el fundamento teórico

Arquitectura

Como ya se ha comentado, DPT tiene por *backbone* un vision transformer (ViT), es decir, fragmenta la imagen original en parches, extrae un *embedding* de cada parche, y pasa este conjunto de vectores como entrada al transformer, de forma similar a como se pasarían los *embeddings* de palabras cuando se aplican estas arquitecturas a texto. El artículo proporciona dos versiones de DPT en función del vision transformer que emplean. La primera tiene por *backbone* una arquitectura ViT large (proyección lineal, x capas,,)

hablar del número de capas de vit large

; y la segunda emplea un ViT Hybrid (que emplea una ResNet50 para extraer el *embedding* de los parches iniciales).

En la parte del *decoder*, se forman representaciones de distintas resoluciones en forma de imagen a partir de los tokens que se encuentran en ciertas capas del transformer. Estas representaciones, se combinan de forma progresiva para obtener la predicción final de la red (la estimación de profundidad para cada píxel).

La operación que convierte los *tokens* de capas intermedias del *transformer* en imágenes se define de la siguiente manera:

Ecuación y probablemente imagen en algún sitio del decoder

1. Read: Integración del readout token: ...
2. Concatenate: ...

3. Resample: ...

Una vez se han extraído estas representaciones intermedias a diferentes resoluciones, se fusionan empleando una arquitectura de tipo RefineNet

citar y explicar la arquitectura

. El resultado de este último bloque, es una imagen con la mitad de resolución que la imagen de entrada que pasa por una cabeza entrenada para proporcionar la predicción final.

Los modelos publicados, han sido entrenados en MIX 6, un dataset elaborado por Intel ampliando MIX 5, también empleado por la empresa en MiDas, el modelo precursor de DPT. Este dataset está compuesto por los siguientes datasets:

Hablar de los dataset de entrenamiento y test, citandoles y haciendo una tabla de número de imágenes, espacio, etc. También se puede mencionar que hay algunos que son difíciles de conseguir debido a las leyes actuales que no se han publicado.

Además de los modelos entrenados en MIX 6, se proporcionan los parámetros de la variante DPT-Hybrid tras realizar un *fine tuning* en los conjuntos de entrenamiento de los datasets NYU Depth v2 y KITTI (cuyos conjuntos de test se emplean posteriormente en este trabajo para comparar los resultados de los distintos modelos resultantes).

Hablar del proceso de entrenamiento que llevan a cabo en el paper de DPT?

Copiar los resultados del paper en ambos datasets y aclarar que esos resultados son los que se presentan en el paper, después en la sección de resultados pondremos los que consigamos nosotros con un proceso de prueba más cercano a lo que sería un entorno de producción, sin dobles predicciones ni cosas similares.

2.9. Optimización de modelos

Los modelos de aprendizaje automático más novedosos, son cada vez más grandes, lo que generalmente conlleva un tiempo de entrenamiento y de inferencia mayor, así como unos 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 inferencia⁵ sin sacrificar la exactitud del modelo en exceso. Siguiendo la clasificación presentada en [62], 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

⁵Dependiendo 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.

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. [63] 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 [64] o en el número medio de ceros que tienen los mapas de características [65] 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)* [66] 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)* [67] 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 [68] como de segundo orden [69].
- **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 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 [70], 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.

2.10. Redes neuronales - Transformers

2.10.1. Transformers para visión

2.11. Estimación de profundidades

2.12. Técnicas de mejora de eficiencia

2.13. Estimación de profundidades eficiente

Empleando (junto a otras) las técnicas de optimización comentadas, se han propuesto distintos modelos que aceleran o fusionan distintas arquitecturas dedicadas a la estimación de profundidades, aparentemente, todos convolucionales. El estudio realizado, si bien es cierto que existen modelos dedicados a la aceleración y reducción de tamaño para hacer estimación de profundidades basados en imágenes de estéreo con aprendizaje supervisado [71] o que emplean imágenes monoculares con aprendizaje no supervisado [72, 73], se centra en aquellos que funcionan sobre imágenes monoculares y cuyo aprendizaje ha sido de tipo supervisado. Los principales modelos con estas características son:

- **FastDepth** [74]: Basado en una arquitectura de tipo *encoder-decoder* convolucional, *FastDepth* emplea como *encoder* la red *MobileNet* [75] y construye el *decoder* con convoluciones separables en profundidad⁶ (*depthwise separable convolutions*) y bloques de *upsample* mediante interpolación de tipo vecino más cercano para aumentar el tamaño del resultado hasta el tamaño de la imagen de entrada. Posteriormente, se poda la red con *NetAdaptV1* [76], un algoritmo capaz de adaptar automáticamente una red al presupuesto de memoria definido. Algo que los autores destacan en el artículo, es cómo normalmente se optimiza solamente el *encoder* en este tipo de arquitecturas, mientras que su propuesta optimiza también el *decoder*.
- **MobileDepth** [77]: De forma similar al modelo anterior, *MobileDepth* se basa en una arquitectura *encoder-decoder* convolucional. En este caso, el *decoder* está compuesto por una red *RegNetY06* [78] mientras que el *decoder* está formado por bloques *split-concatenate shuffle*, inspirados en *ShuffleNet v2* (este tipo de bloque, cuenta con una modificación de las convoluciones separables en profundidad que lo hace ligeramente más rápido). Este modelo, evaluado en NYU Depth v2 [23], presenta mejores resultados que *FastDepth* (Tabla 1), pero es cerca de un 10 % más lento (55ms y 62ms en CPU).

Red	MACs [G]	RMSE	δ_1	CPU [ms]
<i>FastDepth (sin podar)</i>	0.74	0.599	0.775	55
<i>MobileDepth</i>	0.70	0.497	0.827	62

Tabla 1: Resultados comparativos presentados en [77], siendo MAC el número de operaciones de multiplicación y acumulación, RMSE la raíz del error cuadrático medio, δ_1 la exactitud bajo umbral [34] y CPU el tiempo de inferencia.

⁶Partiendo de que las convoluciones separables permiten descomponer el *kernel* en *kernels* de menor dimensión, las convoluciones separables en profundidad, permiten descomponer el *kernel* original por canales, reduciendo drásticamente el número de operaciones. Se puede encontrar más información sobre el funcionamiento de este tipo de convoluciones en [75].

3. Material y métodos

3.1. Software y hardware empleado

Poner links, citas, etc. Repasar todo lo que hay escrito. No usados aún (?): (onnx, kornia, c++)

3.1.1. Software

Citar todo, mencionar las licencias que tiene cada software

Para el desarrollo de este proyecto, se ha optado por el lenguaje de programación **Python 3.7.10** debido a su ecosistema de librerías y código disponible orientado al aprendizaje profundo. Dentro de Python, las librerías y paquetes que se han empleado pueden distinguirse en dos grupos:

- Uso de CPU: **Numpy**, para trabajar con matrices y acelerar operaciones matemáticas; **OpenCV**, para la carga y manipulación de imágenes antes de convertirlas en tensores, y **Matplotlib/Seaborn**, para graficar resultados y otras figuras del documento.
- Uso de GPU: **Pytorch 1.9.0**, 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**, desarrollado y mantenido por Ross Whigtman, que hace disponibles un gran número modelos del estado del arte escritos en Pytorch con sus pesos preentrenados; el **repositorio de DPT**, modelo que se modifica a lo largo del proyecto; y por último, el repositorio **performer-pytorch** de Phil Wang, que ofrece una implementación, también en Pytorch, de la arquitectura Performer y sus capas 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 gracias a su mayor flexibilidad. Consecuencia directa de esto, es que gran parte de los repositorios de código relacionados con publicaciones científicas usan Pytorch para crear sus modelos.

En el parrafo 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 estas tarjetas. El segundo, 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 TODO, respectivamente.

Dada la naturaleza del proyecto, era de esperar que el número de experimentos y variaciones de modelos que se iban a llevar a cabo fuese numeroso, por esta razón, se elige **Weights and Biases (wandb)** para gestionar y monitorizar dichas pruebas, es decir, visualizar y controlar su evolución o registrar métricas y resultados para su posterior comparación. *Weight and Biases* es un servicio de seguimiento de experimentos, gratuito para uso académico y personal, que

se ejecuta en la nube 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 capacidad se ha valorado muy positivamente al compararlo con otros *software* de monitorización como puede ser Tensorboard.

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 manera 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 posteriormente 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 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, y es la portabilidad que ofrece entre sistemas. En caso de querer ejecutar los scripts en cloud (tal y como se explicará en la sección TODO) 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 Anexo TODO.

Por último, para la redacción de este documento se ha empleado **LaTeX** como sistema de composición de texto 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 TODO y TODO respectivamente.

Opción 2, queda más ordenado pero me gusta menos, no creo que sea necesario dedicar un párrafo a cada librería de python.

- **General**

- *Git*:
- **LATEX**: Para redactar este documento, se ha empleado LaTeX como sistema de composición de texto y BibTeX para gestionar las referencias bibliográficas. Además, se ha empleado el editor de código abierto Texmaker.

- **Lenguajes y librerías**

- *Python*
- *NumPy*
- **OpenCV**: Inicialmente desarrollada para C++ pero con *bindings* para Python, esta librería es una herramienta fundamental dentro del campo de la visión artificial, ofreciendo operaciones y transformaciones fundamentales optimizadas para funcionamiento en tiempo real. En este trabajo se ha empleado para llevar a cabo tanto la carga como la manipulación de imágenes antes de convertirlas en tensores.
- **Seaborn**: Construida sobre la librería matplotlib, esta librería proporciona una interfaz de más alto nivel para representar gráficos y visualizaciones. Se ha empleado para obtener algunas de las figuras que aparecen en este documento.

■ Aprendizaje profundo y aceleración por hardware

- **Pytorch**: Dentro del ecosistema de Python, en el momento en el que se ha llevado a cabo este trabajo, existen principalmente dos opciones gratuitas de código abierto para trabajar de forma eficiente con modelos de aprendizaje profundo, Pytorch y Tensorflow/Keras. Estas librerías permiten construir modelos de aprendizaje profundo a partir de funciones y abstracciones que representan distintos tipos de capas, funciones de activación, procesamiento de datos, etc. Ambas librerías permiten la aceleración por *hardware* de estas arquitecturas y su despliegue en sistemas de producción (predominando en el despliegue de aplicaciones Tensorflow). Si hubiese que compararlas, las dos podrían ser perfectamente escogidas para realizar este trabajo de fin de máster. No obstante, Pytorch está tomando la delantera en cuanto a uso en proyectos de investigación. Uno de los proyectos que ha sido desarrollado usando Pytorch es precisamente DPT [5] (arquitectura que se modificará en este proyecto), por lo que para facilitar el desarrollo de las modificaciones a realizar y la reproducción de los resultados de la publicación original se elige dicha opción.
- *timm?* *lucidrains?*:
- *CUDA*, *CuDNN*:
- **Weights and Biases**: Dada la naturaleza del proyecto, era de esperar que los experimentos y entrenamientos llevados a cabo fuesen numerosos. Para poder comparar experimentos, visualizar su evolución y registrar métricas y resultados de una forma efectiva y organizada, se eligió la plataforma *Weight and Biases*. *Weight and Biases* es un servicio de seguimiento de experimentos, gratuito para uso académico y personal, que se ejecuta en la nube y permite registrar de forma sencilla variables y métricas durante los distintos experimentos que se lleven a cabo. Además de esto, ofrece también un gestor de búsqueda de hiperparámetros, también empleado en este trabajo, que gestiona la inicialización de los scripts de entrenamiento en tantas máquinas como se dispongan.

■ Entorno de desarrollo

- **Docker**: Para el desarrollo del proyecto se ha elegido como entorno un contenedor Docker funcionando 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 manera 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 posteriormente se instalan todas las librerías necesarias. 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 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, y es la portabilidad que ofrece entre sistemas. En caso de querer ejecutar los scripts en cloud 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.

- *Pycharm*
- *Jupyter notebook*

(Si se ejecutan las redes en C++ en los entornos embebidos justificarlo), por qué onnx, por qué wandb...

3.1.2. Hardware

Para el desarrollo de este proyecto y entrenamiento/evaluación de los modelos resultantes, se han empleado dos configuraciones de equipos (Tabla 2):

	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 8 GB GDDR6 5888 CUDA cores 184 Tensor cores Arquitectura Ampere	NVIDIA Tesla T4 16 GB GDDR6 2560 CUDA cores 320 Tensor cores Arquitectura Turing
Memoria	32 GB DDR4	15 GB

Tabla 2: Especificaciones de los equipos empleados durante el trabajo de fin de máster.

3.2. Cloud

TODO Repasar, aparece muchas veces instancia, aclarar que imagen es de docker y cual de vm Con el objetivo de reducir el tiempo necesario para entrenar los distintos modelos que se plantean en este trabajo, se ha recurrido al IaaS (*Infrastructure as a Service*) que ofrece la empresa Google. Los servicios en la nube (*cloud*), nos permiten disponer de recursos informáticos de manera flexible, pagando únicamente por aquellos que tengamos 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 crear instancias de máquinas virtuales y **Cloud Storage**, para crear recursos de almacenamiento (*buckets*).

El flujo de trabajo que se ha seguido ha sido el siguiente:

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 instancias y el ordenador local en el que se lleva a cabo el desarrollo (Equipo 1 en Tabla 2).
2. A continuación, se crea una instancia con las características deseadas (Equipo 2 en Tabla 2). Una vez conectados a esta máquina virtual a través de SSH, se copia del *bucket* previamente creado el script de configuración (disponible en el Anexo TODO) 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 de forma que sea fácilmente replicable y 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 de la imagen construida en el cual se ejecuta 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 copia a través de SSH los modelos entrenados que se han guardado en cada una de las instancias empleadas.

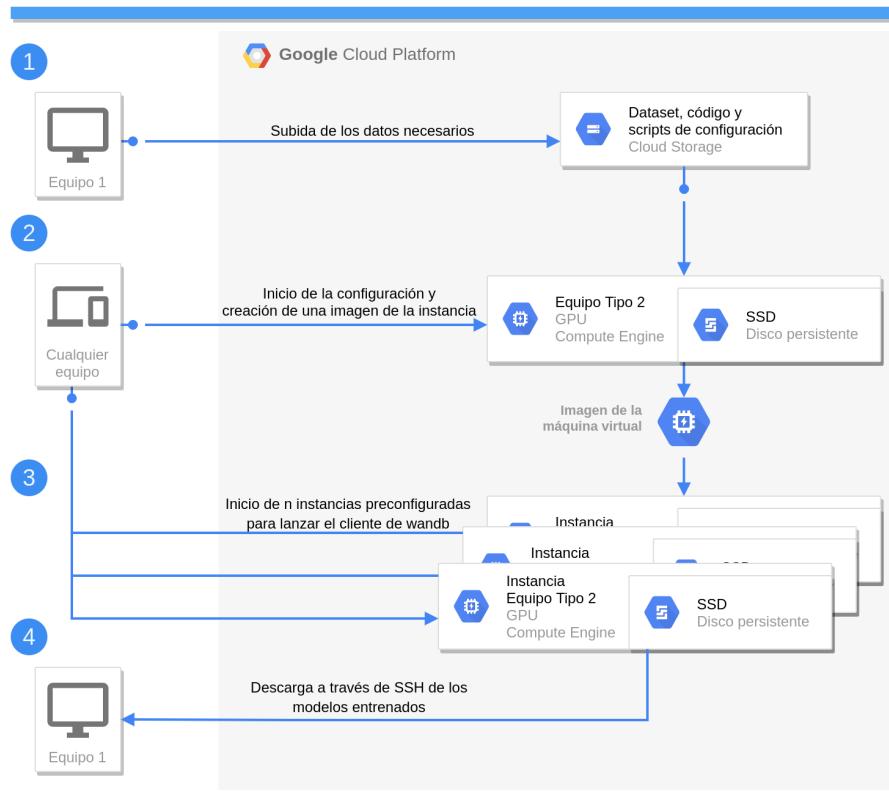


Figura 5: Esquema de la configuración llevada a cabo en la nube.

3.3. Datasets

En esta sección, se introducen aquellos conjuntos de datos, que, de forma directa o indirecta, se emplean en este trabajo.

3.4. ImageNet

ImageNet [6, 7] 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 [8], 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 entrenadas previamente, los modelos han aprendido a extraer características generales (normalmente reutilizables) de las imágenes.

Una muestra de las imágenes que conforman ImageNet está disponible en la Figura 6.

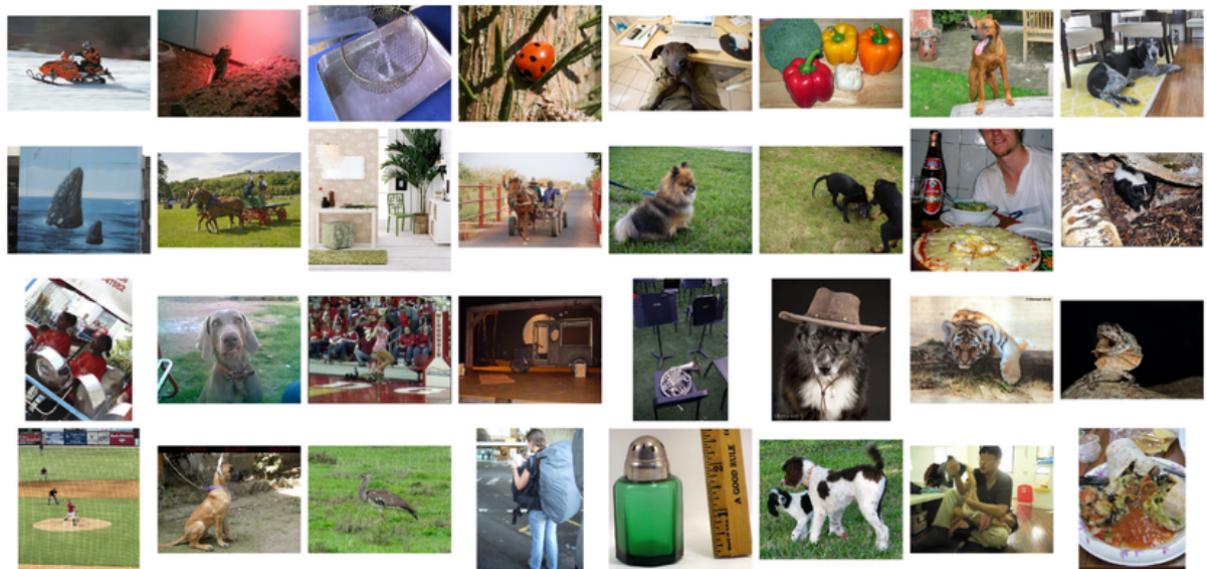


Figura 6: Muestra de las imágenes de ImageNet. Fuente: [7]

3.5. MIX6

MIX6 [5], es una ampliación de MIX5 [9]. Estos dos *datasets*, son en realidad agrupaciones de otros *datasets* que proporcionan anotaciones de profundidad de sus imágenes. Estas agrupaciones consiguen dos puntos importantes: Primero, suman una cantidad de imágenes considerablemente

alta; segundo, al tener los datos 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 ninguna tarea. Estas dos características, hacen que MIX6 sea una muy buena opción especialmente 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 estos *datasets* está disponible en la Tabla 3.

Dataset	Descripción	Núm. de imágenes
Entrenamiento		
DIML Indoor [10]	Imágenes reales anotadas con cámara Kinect de Microsoft.	220K
MegaDepth [11]	Imágenes reales anotadas con MVS (<i>Multi View Stereo</i> - Múltiples puntos de vista en diferentes fotografías)	130K
ReDWeb [12]	Imágenes reales anotadas a partir de estereovisión.	3.6K
WSVD [13]	Vídeos recuperados de YouTube en formato de estereovisión anotados a partir de dicha pareja de imágenes.	1.5M
3D Movies [9]	Películas 3D grabadas con cámaras estereoscópicas anotadas a partir de la pareja de imágenes.	75K
TartanAir [14]	Imágenes sintéticas.	1M
HRWSI [15]	Imágenes reales anotadas a partir de estereovisión.	21K
ApolloScape [16]	Imágenes reales anotadas con sensor LiDAR.	5.1K
BlendedMVS [17]	Imágenes sintéticas.	17K
IRS [18]	Imágenes sintéticas.	103K
Evaluación		
DIW [19]	Imágenes reales anotadas manualmente con la profundidad relativa entre pares de puntos aleatorios.	495K
ETH3D [20]	Imágenes reales anotadas con sensor LiDAR.	5.2K
Sintel [21]	Imágenes sintéticas.	1K
KITTI [22]	Imágenes reales anotadas con sensor LiDAR.	45K
NYUDepthV2 [23]	Imágenes reales anotadas con cámara Kinect de Microsoft.	407K
TUM [24]	Imágenes reales anotadas con cámara Kinect de Microsoft.	87K

Tabla 3: Datasets que conforman MIX6. Subrayados aquellos que no forman parte de MIX5.

3.5.1. KITTI

KITTI [22, 25, 26, 27] 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.

3.5.1.1. Datos disponibles

Los datos disponibles en KITTI fueron capturados empleando un vehículo equipado con diferentes sensores, de especial interés para este trabajo son las dos parejas de cámaras para estereovisión - un montaje con dos cámaras en escala de grises (*2x PointGray Flea2 grayscale cameras, FL2-14S3M-C, 1.4 Megapixels, 1/2" Sony ICX267 CCD, global shutter*) y otro montaje con dos cámaras en color (*2x PointGray Flea2 color cameras (FL2-14S3M-C), 1.4 Megapixels, 1/2" Sony ICX267 CCD, global shutter*) - y un escáner láser *Velodyne HDL-64E rotating 3D laser scanner, 10 Hz, 64 beams, 0.09 degree angular resolution, 2 cm distance accuracy, collecting ~ 1.3 million*

points/second, field of view: 360° horizontal, 26,8° vertical, range: 120 m.

Possiblemente mover los sensores y añadir las lentes del paper a una tabla, incluso subirlo de sección y hablar de todos los sensores que llevaba y ya está

Además de estos sensores, el automóvil también equipaba un sensor *OXTS RT3003* de medida inercial con sistema de navegación GPS para registrar información relacionada con la odometría.

Datos en bruto

Si nos centramos en los datos relevantes para la estimación de profundidades monocular, el *dataset* está compuesto de fotogramas muestreados y sincronizados a 10 Hz de los vídeos capturados por las cámaras previamente descritas en diferentes recorridos. Debido a la naturaleza 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 7.



(a) Imagen en escala de grises capturada por la cámara izquierda.



(b) Imagen en escala de grises capturada por la cámara derecha.



(c) Imagen en color capturada por la cámara izquierda.



(d) Imagen en color capturada por la cámara derecha.

Figura 7: 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 (una cuarta parte de la cantidad original).

Anotaciones

Hablar del formato de las imágenes, rango, etc.

Por otro lado, KITTI proporciona también imágenes formadas por los valores numéricos de la

profundidad para cada uno de los píxeles de las imágenes presentadas previamente. Estos valores son los obtenidos por el escaner láser equipado en el vehículo, y por lo tanto pueden considerarse una medida fiable de la profundidad en cada imagen. Estas imágenes de profundidad serán las que se emplearan como anotaciones y por lo tanto, los valores que se emplearan para entrenar el modelo y evaluar su capacidad de predicción. 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*) y no densas. Esto significa que no todos los píxeles de una imagen dada 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 8. Estas anotaciones están disponibles tanto como para las imágenes capturadas con las cámaras derechas como para las capturadas con las cámaras izquierdas, es decir, hay dos anotaciones para cada instante.

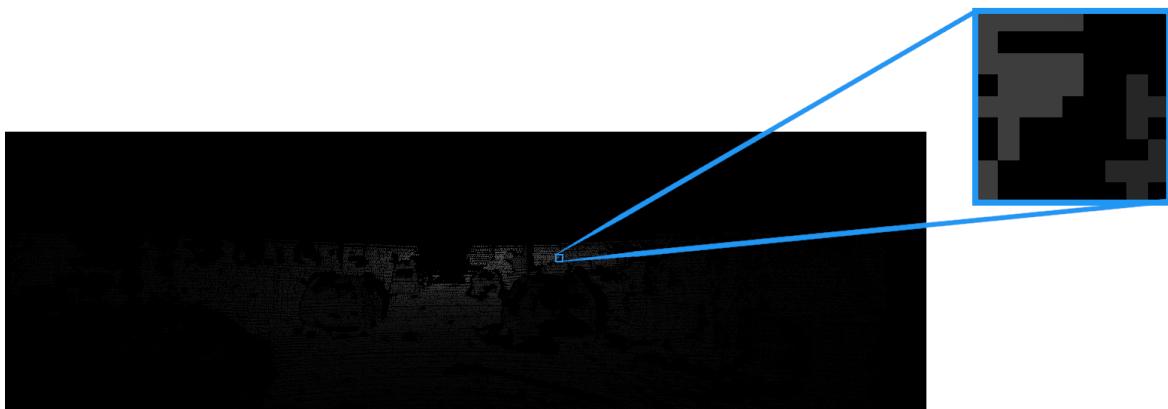


Figura 8: Anotación de KITTI y detalle de su carácter disperso para un instante dado.

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 cada uno 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 científicas sobre estimación de profundidad monocular [?, ?, ?, ?] es común encontrar que se emplean los conjuntos de entrenamiento/validación/evaluación definidos por Eigen et al. [28] (conocido como el *Eigen split*) que no respeta 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 repositorio⁷ del trabajo de Godard et al. [29]. 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 4.

⁷<https://github.com/nianticlabs/monodepth2/tree/master/splits> - (*Eigen full*)

	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 4: 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 encuentras en el conjunto de evaluación.

Adjuntar el código usado para estas cosillas?

3.6. Arquitectura y capas

Hablar de la arquitectura en concreto que se ha utilizado (DPT) apoyándose en todo lo que se haya explicado en el fundamento teórico. Explicar también las capas de atención que se han empleado en más detalle. Apuntar como se hace para estimar la profundidad métrica en un dataset concreto (aquí o **en el apartado de arquitectura**).

3.7. Warmstart

Hablar de que se ha hecho finetuning en vez de entrenar desde cero por las limitaciones de computo y de disponibilidad de datos. Aunque no todas las capas coincidiesen, se han empleado los pesos entrenados en general, NO los finetuneados ya en sus respectivos datasets (Esto en realidad estaría bien probarlo a ver si no hace mucho overfitting).

3.8. Función de pérdida

Hablar de la función de pérdida empleada para entrenar, hay que ver como se puede hilar esto con una sección en el fundamento teórico donde se expliquen más funciones de pérdida para estimación de profundidades.

3.9. Data augmentation

Hablar de data augmentation si se lleva a cabo, exponer las operaciones y justificarlas con regularización y generalización a casos nuevos, evitar que siempre se itere sobre las mismas imágenes.

3.10. 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. [30], que también es la empleada para evaluar los resultados del *Dense Prediction Transformer* [5]. De esta forma, se satisfacen dos objetivos: reproducir los resultados presentados en dicho artículo con sus modelos, y evaluar de una forma justa las modificaciones introducidas. Un punto importante de esta evaluación es que si bien es cierto que las modificaciones introducidas en este trabajo reducen el tamaño de la imagen en la entrada de las arquitecturas, el resultado se escala a su tamaño original antes de llevar a cabo la evaluación, asegurando de esta forma la consistencia en la evaluación de las distintas variaciones realizadas en la arquitectura. Los pasos a seguir son:

Recortes

Masks in zones with no info

3.10.1. Métricas

Una vez recortadas 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 monocularas [5, 9, 28, 30, 31, 32, 33, 34].

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, ya que como se ha comentado previamente, no todas las anotaciones tienen información disponible para todos los píxeles de la imagen (anotaciones dispersas).

3.10.1.1. Accuracy under a threshold

La primera de estas métricas, el *accuracy under a threshold*, viene dada por la Ecuación 1 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.

$$\% \text{ de } p \in T : \max\left(\frac{\hat{d}_p}{d_p}, \frac{d_p}{\hat{d}_p}\right) = \delta < \text{umbral} \quad (1)$$

3.10.1.2. 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 2).

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

3.10.1.3. 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). De esta forma, por la naturaleza cuadrática de la fórmula, se le da una mayor importancia a los errores mayores que a los menores.

$$\frac{1}{T} \sum_{p \in T} \frac{(d_p - \hat{d}_p)^2}{d_p} \quad (3)$$

3.10.1.4. 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 4). Dos características interesantes de esta métrica son que su valor se puede interpretar

como la desviación estándar de la varianza residual y 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.

revisar
esto

$$\sqrt{\frac{1}{T} \sum_{p \in T} (d_p - \hat{d}_p)^2} \quad (4)$$

3.10.1.5. 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 5). 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, obteniendo así 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.

$$\sqrt{\frac{1}{T} \sum_{p \in T} (\ln d_p - \ln \hat{d}_p)^2} \quad (5)$$

3.10.1.6. Scale Invariant Logarithmic Error (*SILog*)

Esta métrica, es una versión modificada de la función de pérdida propuesta por Eigen et al. obtenida fijando el valor de $\lambda = 1$, calculando su raíz cuadrada, y multiplicando finalmente por 100 (Ecuación 6). Al fija el valor de λ en la unidad, se obtiene una medida totalmente independiente de la escala de la salida. 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 se ha comentado anteriormente, es uno de los problemas fundamentales de la estimación de profundidades en imagen monocular.

$$\sqrt{\frac{1}{T} \sum_{p \in T} (\ln d_p - \ln \hat{d}_p)^2 - \left(\frac{1}{T} \sum_{p \in T} \ln d_p - \ln \hat{d}_p \right)^2} * 100 \quad (6)$$

3.10.1.7. 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 7.

$$\frac{1}{T} \sum_{p \in T} |\log_{10} d_p - \log_{10} \hat{d}_p| \quad (7)$$

citar a eigen y hacer una referencia al apartado de funciones de pérdida cuando esté hecho. o justificar por qué es invariante a la escala o referenciar la explicación de la función de pérdidas donde se explique

3.10.1.8. 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 en este trabajo, hay dos grupos, aquellas condicionadas por el *hardware* en el que se realizan las pruebas (Tiempo de inferencia y Tasa de transferencia efectiva) y aquellas independientes de este (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.

Tasa de transferencia efectiva (*Throughput*)

Por otro lado, en caso de que el procesamiento de imágenes se haga de forma offline y se disponga de todas las imágenes antes de comenzar el procesamiento, estas se podrían agrupar en lotes (*batches*) para parallelizar su inferencia. Al parallelizar el procesamiento de las entradas, aumenta el número de imágenes que se puede procesar por unidad de tiempo, que es lo que medirá esta métrica. Es decir, la tasa de transferencia efectiva es el número máximo de imágenes que puede procesar un modelo por unidad de tiempo. De nuevo, como se ha mencionado en el párrafo introductorio, este valor está ligado al equipo en el que se lleve a cabo la inferencia.

Número de operaciones en coma flotante (*FLOPs*)

Número de FLOPs que el modelo tiene que llevar a cabo para procesar una sola entrada.

O desarrollar más esto o hablar de MACs, o hablar de ambas

Si quantizamos los modelos a int8 dejamos de tener operaciones en coma flotante y esta métrica no servirá de nada. Explorar la opción de usar MACs (https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation). En el paper de FastDepth es lo que hacen.

3.11. Portabilidad (?) de los modelos

Explicar el proceso que se ha llevado a cabo con onnx y por qué se emplea, explicar que hace onnx por debajo, hacer diagramas. Puede que esto colapse con la sección de software de arriba, se puede quitar.

4. Modificaciones de la arquitectura y desarrollo

MEter los resultados y las imágenes aquí? O en la sección de resultados? Podriamos o hacer referencia a esta sección en la de discusión o limitar esta sección a esquemas, código y explicación teórica de que se ha cambiado para luego en la sección de resultados repetir entre comillas los apartados de esta sección solamente con las gráficas pertinentes (evidentemente comentandolas). Puede que sea mejor opción la segunda

4.1. Entrenamiento

Script de entrenamiento, dataloader, todas las cosas que se han hecho para acelerar el entrenamiento. Hablar de las horas de entrenamiento? Puede que se mezclen cosas con el apartado de metodología?

4.2. Reducción de tamaño de la entrada

Para acelerar DPT, lo primero que se modificó fue el tamaño de la entrada. 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. La primera operación emplea como método de interpolación el algoritmo `INTER_AREA` de OpenCV, que TODO y está recomendado para reducir el tamaño de imágenes ya que proporciona resultados sin efecto Moire; la operación que amplia la salida al tamaño de la imagen original, por otro lado, emplea interpolación bicúbica, que ofrece los mejores resultados pese a ser más lenta, ya que el tiempo empleado en las operaciones de reescalado es despreciable en comparación con el tiempo de inferencia de la red.

Medir el tiempo que está escalando y haciendo inferencia para justificar la frase del final

Hacer una comparación de velocidad de inferencia con la imagen en grande y con la imagen en pequeño

Entrenar un modelo con la imagen grande para ver como afecta? Podría ser solo uno, similar a la prueba que se quiere hacer con el resnet entrenado en imagenet

Explicar `inter_area`, citarlo? explicar en una nota al pie qué es el efecto moire?

4.3. Número de cabezas

Hablar del cambio en el número de cabezas

Citar y justificar esta dirección de búsqueda con el paper de are 16 heads better than 1

4.4. Capas de atención eficiente

Cambio de las capas, hacer una gráfica midiendo en función del tamaño de la cadena la velocidad en la que pasa por una de estas capas? Puede ser interesante. (Sería para imágenes mayores)

Hacer un estudio del incremento de velocidad en función del tamaño de los tokens, es posible que haga falta usar una máquina de Google Cloud para esto, se puede coger una gorda con mucha memoria, habría que incluirla en el apartado de hardware, también se podría usar para el cálculo de el flujo máximo de imágenes (una A100/P100). Se podría usar la misma imagen en teoría, si sale bien mencionarlo también en el apartado de Docker o en el de Google Cloud

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

Hablar del cambio en los hooks, en el paper original se valora el cambio de hooks en la etapa convolucional, pero no en el transformer, se estudian 0,1; 2,5; 8,11

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. Esta modificación, además de acelerar el entrenamiento e inferencia del modelo, 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 9, 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 pueden eliminarse ya que la inferencia del modelo solo llega hasta este segundo bloque.

Hacer una comparación del tamaño del modelo cuando se eliminan los pesos

Hacer una comparación de los resultados en validación en función de los hooks

Hacer una comparación de la velocidad de inferencia en función de los hooks empleados

Mencionar en la discusión que esto está muy bien porque no están reduciendo la complejidad de la atención, te la estás cargando directamente

Meter el código aquí o una referencia a las partes del código donde se encuentra este cambio?

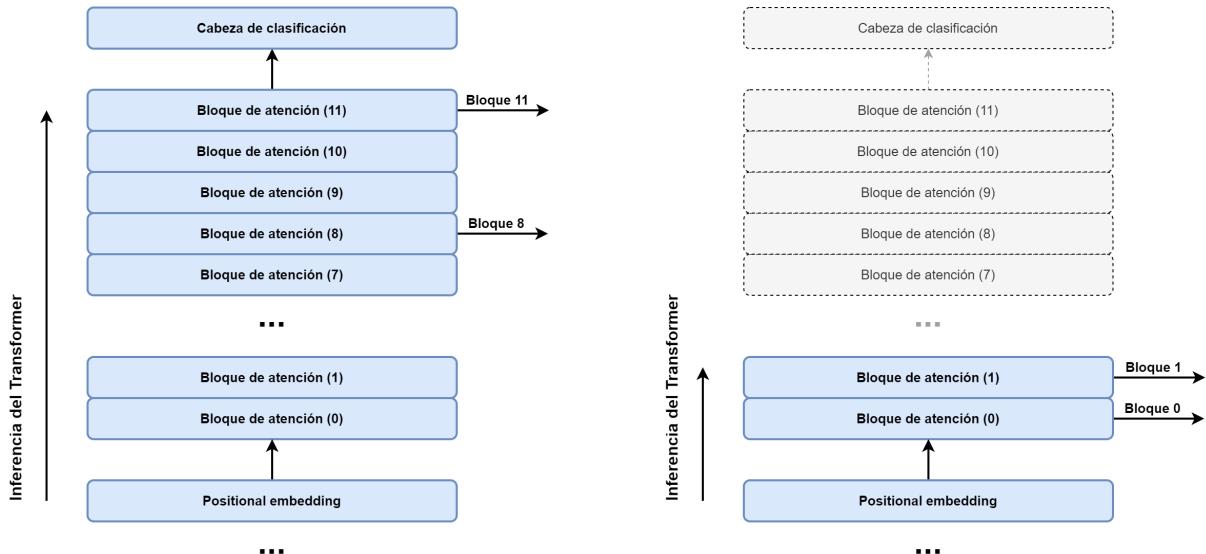


Figura 9: Cambio en el número de bloques de atención.

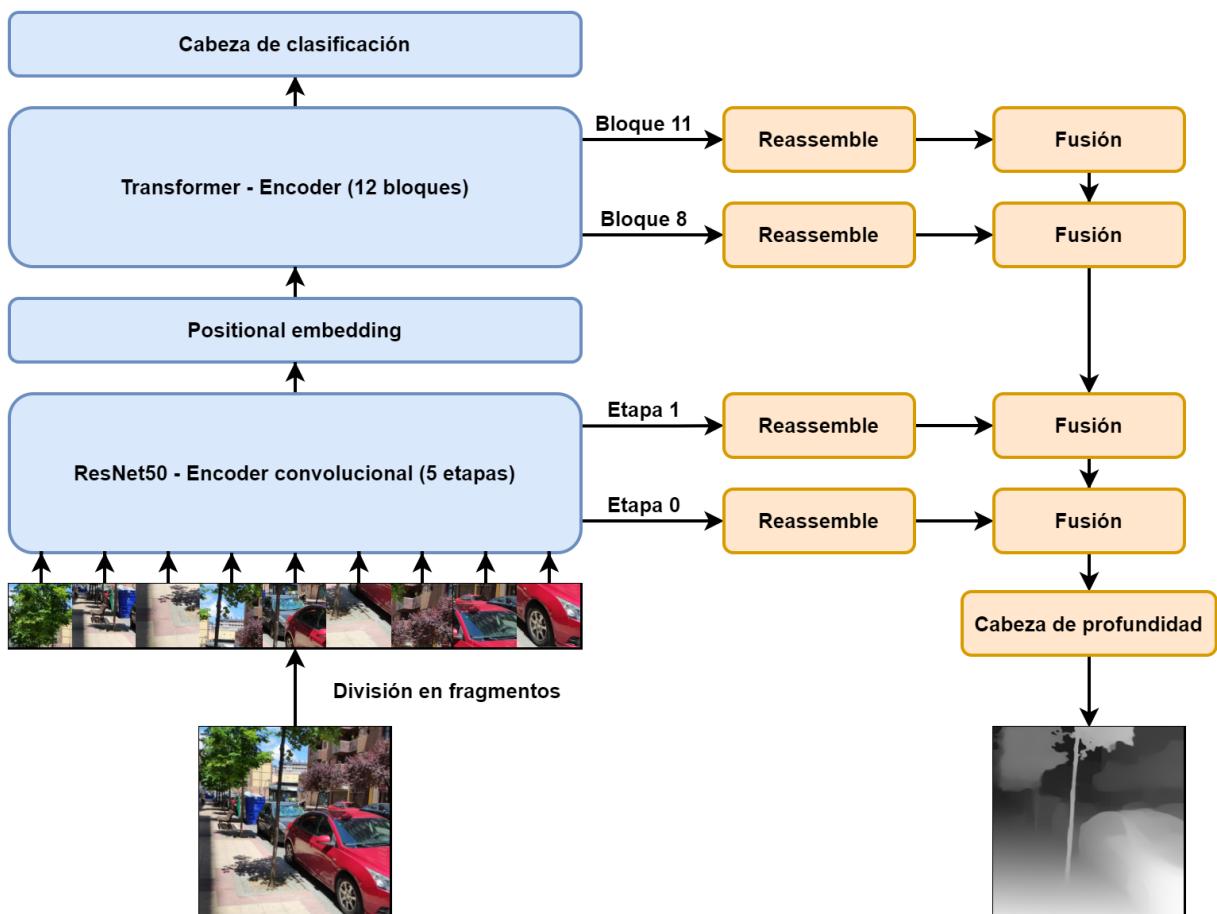


Figura 10: Arquitectura general DPT.

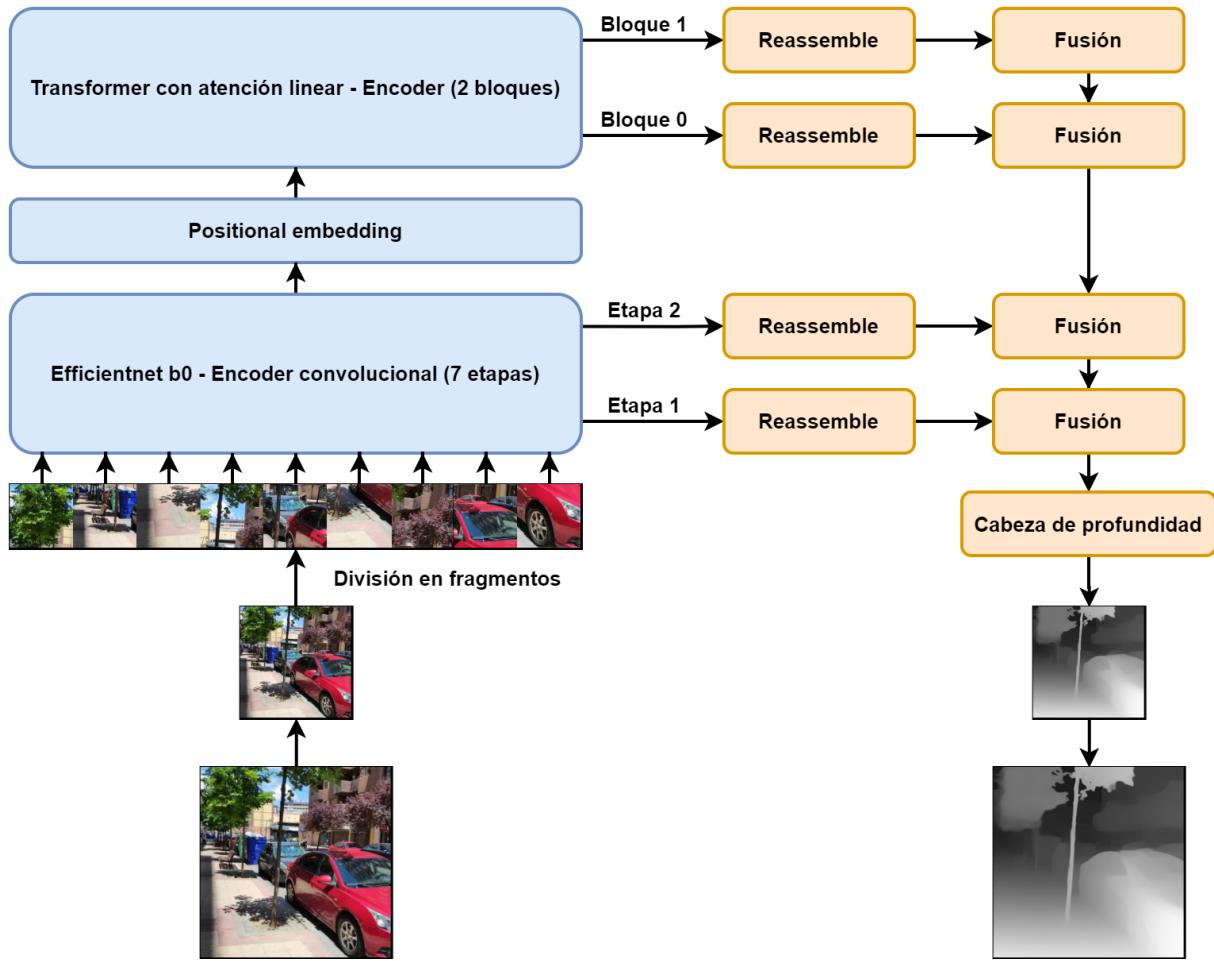


Figura 11: Arquitectura general DPT tras las modificaciones.

4.6. Cambio del backbone convolucional

El último cambio estudiado en este proyecto es el del backbone convolucional del Hybrid ViT. En el modulo propuesto en la publicación original, se elige como backbone una ResNet50v2, de la que se extraen las activaciones en los bloques 0 y 1, que tienen un tamaño TODO de 256 y 512, quedando así TODO de tamaños [256, 512, 768, 768]. Por otro lado, la salida de la última capa de la ResNet50v2, es decir, la entrada de los bloques de atención, tiene forma $[n, c, h, w]$, donde n es el número de imágenes en el batch (batch size), c el número de canales (en este caso mapas de características), que son 1024 por como está diseñada la arquitectura, y por último, h y w , que son la altura y anchura de los mapas de características y son iguales a las dimensiones 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 1x1 y stride 1x1, con 1024 canales de entrada y 768 canales de salida. Este tipo de capas, realmente son 786 kernels de 1x1x1024, por lo que al convolucionar cada uno de ellos la entrada, se obtienen 768 mapas de características del mismo tamaño que los de la entrada. Los 768 mapas de características resultantes, se apllanan en tensores de forma $[n, 768, h/8 * w/8]$ 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, pero la dimensión de estos permanece constante.

Hablar de las convoluciones con kernels 1x1 arriba en vez de aquí?

A modo de ejemplo, supongamos una entrada de una sola imagen de tamaño 384x384: la salida

de la ResNet50v2 tendrá la forma [1, 1024, 24, 24]. Esta salida atraviesa la capa de proyección y pasa a tener forma [1, 768, 24, 24]. Estos mapas de características se aplanan en un tensor [1, 768, 576] que se traspone para obtener la forma [1, 576, 768], que equivale a 576 tokems de dimensión 768. Una vez llegados a este punto, el tensor está listo para pasar a los bloques de atención del transformer.

Por otro lado, la arquitectura EfficientNet-B0, alternativa usada en las pruebas de este trabajo, proporciona una salida (antes de la capa de global pooling) de forma [n, c, h, w] donde n vuelve a ser el número de imágenes procesadas en paralelo, el número de mapas de características es en este caso 1280, y la altura y anchura se corresponden con las de las imágenes de entrada divididas 5 veces entre 2. Es decir: [n, 1280, h/16, w/16]. Para poder sustituir el backbone convolucional del Hybrid ViT sin tener que cambiar el tamaño de todas las capas de atención (para poder 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 1x1 por una capa de convolución transpuesta. Esta capa de convolución transpuesta, cumple dos funciones fundamentales: la primera, transforma los 1280 mapas de características en 768; y la segunda, al tener un kernel de tamaño 2x2 y un stride también de 2x2, consigue que las dimensiones de estos mapas de características se multipliquen exactamente por 2 en ambas dimensiones, convirtiendo la entrada [n, 1280, h/16, w/16] en [n, 1280, h/8, w/8], que es lo que espera la etapa que aplana los mapas de características y transpone el tensor para obtener de nuevo la entrada preparada para los bloques de atención de forma [n, t, 768] donde t vuelve a ser el número de tokems que pasan al transformer, cada uno de dimensión 768.

Explicar bien y añadir en el parrafo de efficientnet el tamaño que tienen los hooks de la red convolucional

Meter alguna imagen para explicar todo este apartado

Decir lo de que se ha modificado la carga de pesos del modelo

Decir (y luego repetir) que efficientnet no está preentrenada en mix y que esto evidentemente afectará a los resultados

5. Resultados

En este capítulo se exponen los resultados obtenidos tras entrenar las arquitecturas con las modificaciones planteadas. Para poder hacer un estudio riguroso, se han tomado una serie de valores candidatos para cada modificación (contarles) y se ha llevado a cabo el producto cartesiano de dichos conjuntos, obteniendo así $3 \times 2 \times 3 \times 2 = 36$ distintos modelos.

Hablar de resultados cuantitativos (tablas con las métricas en distintas pruebas) y cualitativos (imágenes resultado de cada una de las pruebas - escaladas si es necesario - con el ground truth y con la imagen real de referencia). Mencionar que para que se vean mejor las imágenes los valores se han hecho relativos para así ocupar toda la escala de grises.

Meter comparación de los tiempos de inferencia con autocast y sin autocast

5.1. Resultados cuantitativos

5.1.1. Reducción de tamaño de la entrada

Dado que el entrenamiento de los modelos - por motivos de disponibilidad de recursos físicos y temporales - se ha llevado a cabo con la entrada del modelo reducida, no es posible realizar una comparación exhaustiva de los resultados obtenidos para cada una de las modificaciones de la arquitectura respecto del modelo correspondiente entrenado con las imágenes en su tamaño original.

Hacer una prueba si cabe en la GPU entrenando solo un modelo

Poner las dos barras del mismo color?

En cuanto a la medida de la velocidad de inferencia, que si que es posible obtener independientemente de que los pesos correspondan al modelo entrenado o no, en la Figura 12 se puede apreciar la diferencia en función del tamaño de entrada de los FPS medios 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 [?], 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 representan los FPS en función de los otros cambios mencionados en la sección TODO se presentará la influencia de dichos cambios en la inferencia tanto con la entrada original como con la entrada reducida para poder así valorar también la mejora de rendimiento que conllevaría la modificación si no se hubiese modificado el tamaño de la entrada.

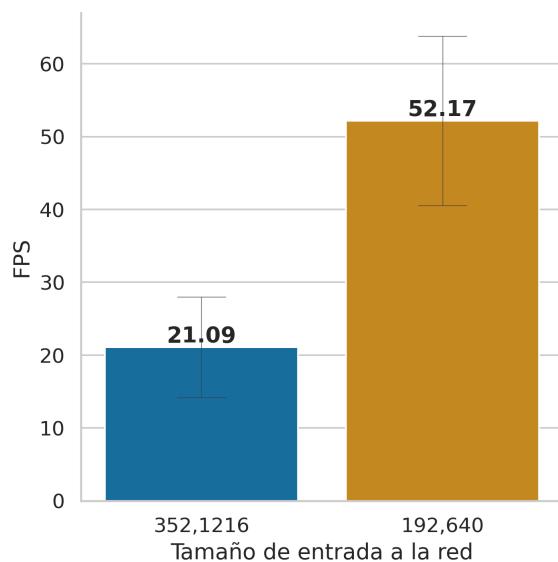


Figura 12: FPS promedios de los modelos en función del tamaño de su entrada. Las barras grises representan la desviación estándar de las medidas.

5.1.2. Número de cabezas

Completar el párrafo de abajo cuando estén los resultados de las otras métricas, a lo mejor algo de la explicación de los resultados llevarlo a un apartado de discusión?

Respecto de la influencia de este parámetro en la velocidad de inferencia, en la Figura 13 se puede observar, agrupado por tamaño de entrada, backbone, y número de cabezas, los promedios de los FPS de los modelos. Teniendo en cuenta que las capas de atención del modelo original tienen 12 cabezas, era de esperar que aquellos modelos con una sola cabeza fueran más rápidos y los que cuentan con 24 fuesen más lentos. Además, este cambio en la velocidad de inferencia relativa es mayor en el caso de la entrada sin escalar que en el caso de la entrada escalada. Por ejemplo, en el paso de 12 a una cabeza con la entrada sin escalar, el incremento en el número de FPS es de $\times 1,2$ y $\times 1,26$ dependiendo del backbone empleado, mientras que al escalar la salida este cambio pasa a ser $\times 1,04$ y $\times 1,03$ respectivamente. Esto se debe a la implementación previamente comentada en la sección TODO, ya que TODO reescribir esto

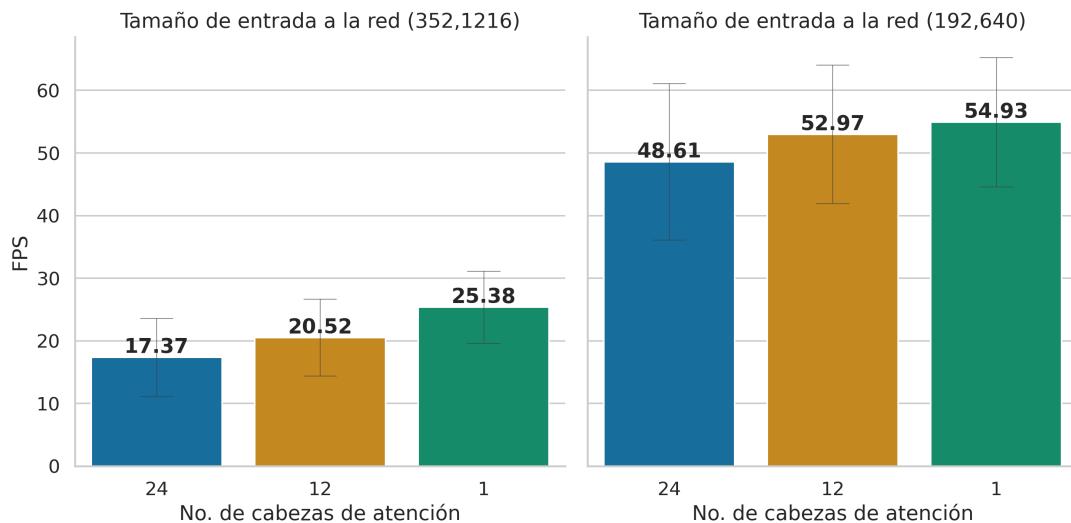


Figura 13: Resultados de.

5.1.3. Capas de atención eficiente

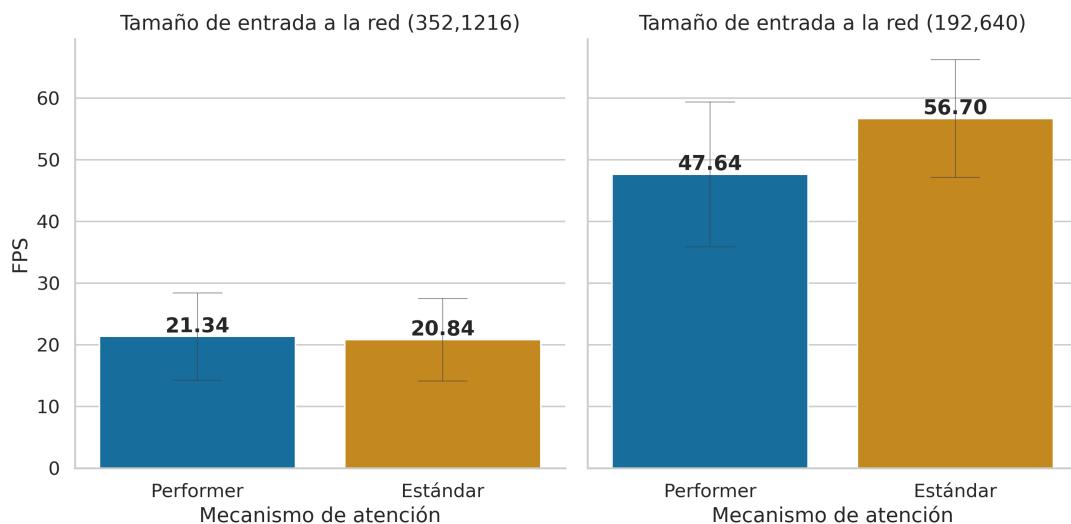


Figura 14: Resultados de.

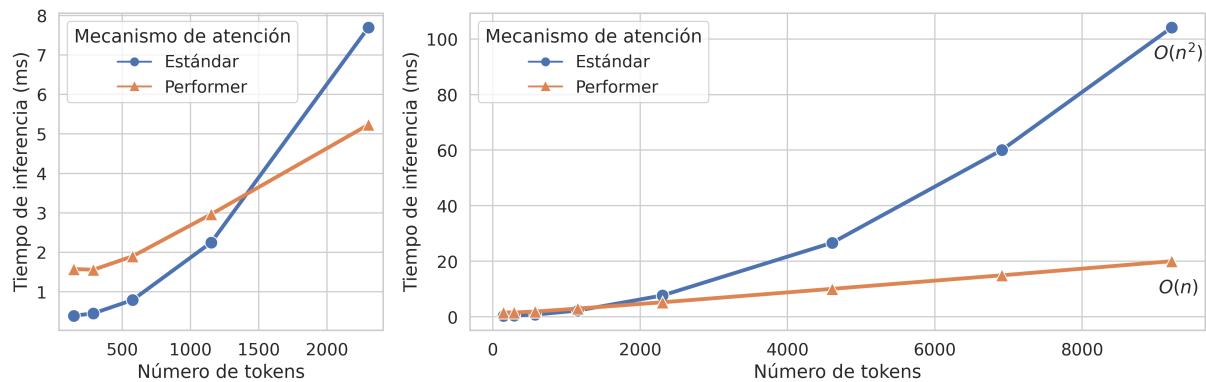


Figura 15: Resultados de.

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

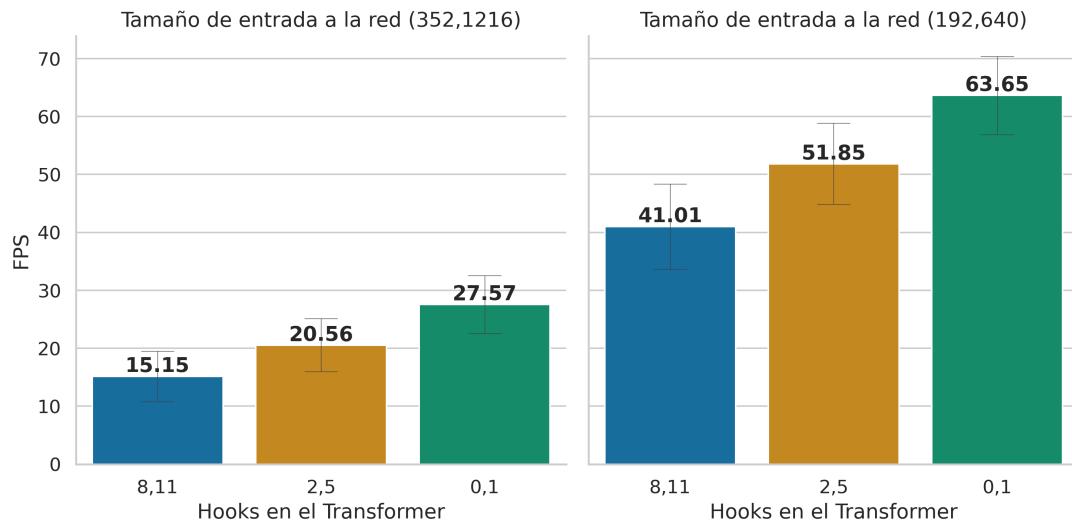


Figura 16: Resultados de.

5.1.5. Cambio del backbone convolucional

Comparar los feature maps que saca uno y otro?

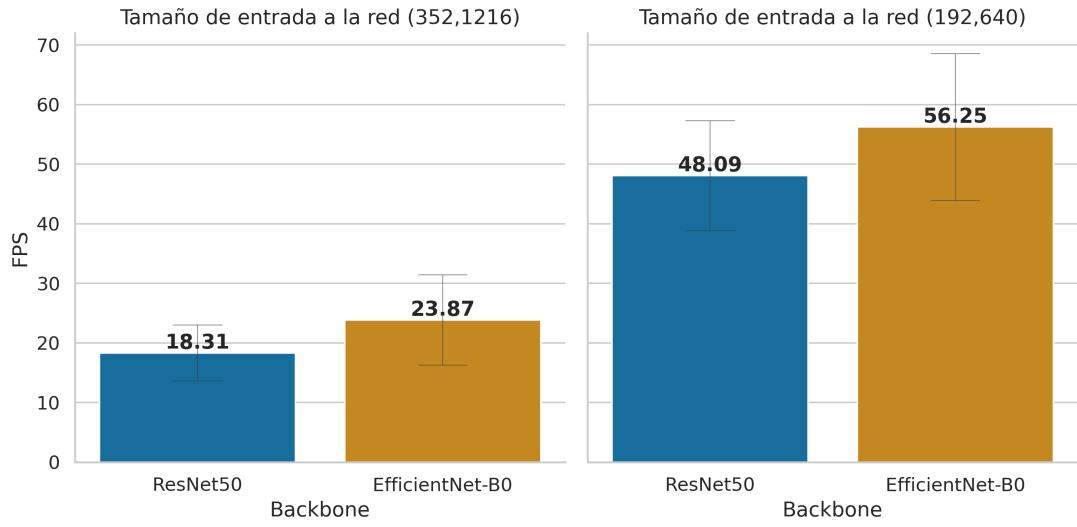


Figura 17: Resultados de.

5.2. Resultados cualitativos



Figura 18: Comparación cualitativa de resultados en el conjunto de evaluación entre el modelo DPT original y el modelo DPT con las modificaciones desarrolladas en este trabajo. Para facilitar su visualización, el rango de las profundidades predichas se ha ajustado a toda la escala de grises.

6. Discusión

Hablar y comentar los resultados obtenidos con otros resultados (convolucionales? y transformers), si los resultados en la jetson son lentos y no se puede acceder a otra más potente, buscar referencias para ver el incremento de rendimiento que presentan con otros algoritmos para poder dar una estimación de cuanto más rápido podría ir en otro hardware embebido más potente.

Mencionar que las capas eficientes no afectan muchisimo debido a que el tamaño de las cadenas de tokens no son demasiado grandes (estamos trabajando con imágenes pequeñas y no con larguisimas cadenas de texto para las que fueron diseñadas).

7. Conclusiones y líneas futuras

Overview del trabajo y de los resultados obtenidos, hacer hincapié en los modelos producidos y su utilidad, hablar del interés de la gente en optimizar este tipo de modelos (github)...

Decir que queda pendiente probar en otro hardware empotrado, entrenar con más datos, reentrenar desde cero sin hacer warmstart, entrenar en imagenet antes las transformers con las capas de atención modificadas, explorar más técnicas de optimización que no se han empleado, más atenciones eficientes, aplicarlo a imágenes mucho mayores. Por otro lado, desarrollar aplicaciones con los modelos producidos como hacer un slam monocular, conducción autónoma,

Bibliografía

- [1] R. Hartley and A. Zisserman, *Two-View Geometry*, p. 237–238. Cambridge University Press, 2 ed., 2004.
- [2] M. Kalloniatis and C. Luu, *Webvision: The Organization of the Retina and Visual System*. 2005.
- [3] 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.
- [4] D. Zhang, S. Mishra, E. Brynjolfsson, J. Etchemendy, D. Ganguli, B. Grosz, T. Lyons, J. Manyika, J. C. Niebles, M. Sellitto, Y. Shoham, J. Clark, and R. Perrault, “The AI Index 2021 Annual Report,” tech. rep., AI Index Steering Committee, Human-Centered AI Institute, Stanford University, Stanford, CA, Mar. 2021.
- [5] R. Ranftl, A. Bochkovskiy, and V. Koltun, “Vision transformers for dense prediction,” 2021.
- [6] 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.
- [7] 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.
- [8] C. Fellbaum, ed., *WordNet: An Electronic Lexical Database*. Language, Speech, and Communication, Cambridge, MA: MIT Press, 1998.
- [9] 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.
- [10] 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.
- [11] Z. Li and N. Snavely, “Megadepth: Learning single-view depth prediction from internet photos,” in *Computer Vision and Pattern Recognition (CVPR)*, 2018.
- [12] 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.
- [13] C. Wang, S. Lucey, F. Perazzi, and O. Wang, “Web stereo video supervision for depth prediction from dynamic scenes,” 2019.
- [14] 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.
- [15] 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.
- [16] 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.

- [17] 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.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, “Vision meets robotics: The kitti dataset,” *International Journal of Robotics Research (IJRR)*, 2013.
- [23] P. K. Nathan Silberman, Derek Hoiem and R. Fergus, “Indoor segmentation and support inference from rgbd images,” in *ECCV*, 2012.
- [24] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, “A benchmark for the evaluation of rgb-d slam systems,” in *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.
- [25] 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.
- [26] 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.
- [27] M. Menze and A. Geiger, “Object scene flow for autonomous vehicles,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] 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.

- [32] S. F. Bhat, I. Alhashim, and P. Wonka, “Adabins: Depth estimation using adaptive bins,” 2020.
- [33] T. Koch, L. Liebel, F. Fraundorfer, and M. Korner, “Evaluation of cnn-based single-image depth estimation methods,” in *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, September 2018.
- [34] 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.
- [35] 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.
- [36] 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.
- [37] 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.
- [38] 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.
- [39] 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.
- [40] 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.
- [41] 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.
- [42] 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.
- [43] 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.
- [44] 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.

- [45] T. Zhou, M. Brown, N. Snavely, and D. G. Lowe, “Unsupervised learning of depth and ego-motion from video,” in *CVPR*, 2017.
- [46] S. Vijayanarasimhan, S. Ricco, C. Schmid, R. Sukthankar, and K. Fragkiadaki, “Sfm-net: Learning of structure and motion from video,” 2017.
- [47] 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.
- [48] 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.
- [49] 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.
- [50] 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.
- [51] 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.
- [52] 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.
- [53] J. Xie, R. B. Girshick, and A. Farhadi, “Deep3d: Fully automatic 2d-to-3d video conversion with deep convolutional neural networks,” in *ECCV*, 2016.
- [54] 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.
- [55] F. Ma, G. V. Cavalheiro, and S. Karaman, “Self-supervised sparse-to-dense: Self-supervised depth completion from lidar and monocular camera,” 2019.
- [56] M. Hu, S. Wang, B. Li, S. Ning, L. Fan, and X. Gong, “Towards precise and efficient image guided depth completion,” 2021.
- [57] 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.
- [58] 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.
- [59] L. Zwald and S. Lambert-Lacroix, “The berhu penalty and the grouped effect,” 2012.
- [60] 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.

- [61] 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.
- [62] S. Vadera and S. Ameen, “Methods for pruning deep neural networks,” 2020.
- [63] 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.
- [64] A. Polyak and L. Wolf, “Channel-level acceleration of deep face representations,” *IEEE Access*, vol. 3, pp. 2163–2175, 2015.
- [65] 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.
- [66] 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.
- [67] 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.
- [68] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, “Pruning convolutional neural networks for resource efficient inference,” 2017.
- [69] 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.
- [70] 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.
- [71] C. A. Aguilera, C. Aguilera, C. A. Navarro, and A. D. Sappa, “Fast cnn stereo depth estimation through embedded gpu devices,” *Sensors*, vol. 20, no. 11, 2020.
- [72] M. Poggi, F. Aleotti, F. Tosi, and S. Mattoccia, “Towards real-time unsupervised monocular depth estimation on cpu,” in *IEEE/JRS Conference on Intelligent Robots and Systems (IROS)*, 2018.
- [73] J. Liu, Q. Li, R. Cao, W. Tang, and G. Qiu, “Mininet: An extremely lightweight convolutional neural network for real-time unsupervised monocular depth estimation,” *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 166, pp. 255–267, 08 2020.
- [74] Wofk, Diana and Ma, Fangchang and Yang, Tien-Ju and Karaman, Sertac and Sze, Vivienne, “FastDepth: Fast Monocular Depth Estimation on Embedded Systems,” in *IEEE International Conference on Robotics and Automation (ICRA)*, 2019.
- [75] A. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” 04 2017.
- [76] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, “Netadapt: Platform-aware neural network adaptation for mobile applications,” in *The European Conference on Computer Vision (ECCV)*, September 2018.

- [77] Y. Wang, “Mobiledepth: Efficient monocular depth prediction on mobile devices,” 2020.
- [78] I. Radosavovic, R. Kosaraju, R. Girshick, K. He, and P. Dollar, “Designing network design spaces,” in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, (Los Alamitos, CA, USA), pp. 10425–10433, IEEE Computer Society, jun 2020.
- [79] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, “Efficient Transformers: A Survey,” *arXiv e-prints*, p. arXiv:2009.06732, Sept. 2020.
- [80] 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.
- [81] 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.
- [82] R. Child, S. Gray, A. Radford, and I. Sutskever, “Generating Long Sequences with Sparse Transformers,” *arXiv e-prints*, p. arXiv:1904.10509, Apr. 2019.
- [83] I. Beltagy, M. E. Peters, and A. Cohan, “Longformer: The long-document transformer,” 2020.
- [84] 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.
- [85] N. Kitaev, L. Kaiser, and A. Levskaya, “Reformer: The efficient transformer,” in *International Conference on Learning Representations*, 2020.
- [86] 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.
- [87] Q. Zhang and Y. Yang, “Rest: An efficient transformer for visual recognition,” *arXiv preprint arXiv:2105.13677v2*, 2021.
- [88] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, “Linerformer: Self-attention with linear complexity,” 2020.
- [89] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, “Transformers are RNNs: Fast autoregressive transformers with linear attention,” in *Proceedings of the 37th International Conference on Machine Learning* (H. D. III and A. Singh, eds.), vol. 119 of *Proceedings of Machine Learning Research*, pp. 5156–5165, PMLR, 13–18 Jul 2020.